NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

ALLOY EXPERIMENTS FOR A LEAST PRIVILEGE SEPARATION KERNEL

by

David A. Phelps

June 2007

Thesis Advisor:	Mikhail Auguston
Co-Advisor:	Timothy Levin

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2007	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE: Alloy Experiments for a Least Privilege Separation Kernel	**5. FUNDING NUMBERS**
6. AUTHOR(S) Phelps, David A.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

A least privilege separation kernel (LPSK) is part of a long-term project known as the Trusted Computing Exemplar (TCX). A major objective of the TCX is the creation of an open framework for high assurance development. A relatively new specification tool called Alloy has shown potential for high assurance development. We implemented the formal security policy model (FSPM) and the formal top level specification (FTLS) of the TCX LPSK in Alloy and concluded that Alloy has few limitations and is more than sufficiently useful, as measured by utility and ease of use, to include in the TCX framework.

14. SUBJECT TERMS Formal Methods, High Assurance Systems, Separation Kernel, Alloy, Morphisms, Model Checker, TCX, LPSK	**15. NUMBER OF PAGES** 107
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

THIS PAGE INTENTIONALLY LEFT BLANK

ALLOY EXPERIMENTS FOR A LEAST PRIVILEGE SEPARATION KERNEL

David A. Phelps
Civilian, United States
B.S., Portland State University, 2005

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

**NAVAL POSTGRADUATE SCHOOL
June 2006**

Author: David A. Phelps

Approved by: Mikhail Auguston
Thesis Advisor

Timothy Levin
Co-Advisor

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A least privilege separation kernel (LPSK) is part of a long-term project known as the Trusted Computing Exemplar (TCX). A major objective of the TCX is the creation of an open framework for high assurance development. A relatively new specification tool called Alloy has shown potential for high assurance development. We implemented the formal security policy model (FSPM) and the formal top level specification (FTLS) of the TCX LPSK in Alloy and concluded that Alloy has few limitations and is more than sufficiently useful, as measured by utility and ease of use, to include in the TCX framework.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to give a tremendous thanks to my thesis advisors Mikhail Auguston and Timothy Levin. Their patience with me and constant encouragement of me did not go unnoticed and was greatly appreciated. I would like to thank Cynthia Irvine and her support of the Scholarship For Service program at the Naval Postgraduate School. Lastly, I would like to thank my wife who carried and bore our first child while I worked on this thesis.

This material is based on work supported by the National Science Foundation under Grant No. DUE—0414102. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

EXECUTIVE SUMMARY

A least privilege model for separation kernels (LPSK) is an example of a high assurance software system and a key component of a long term development project of the CISR group at the Naval Postgraduate School (NPS) known as the Trusted Computing Exemplar (TCX). An overarching goal of the TCX project is the establishment of an open framework for rapid high assurance development [Irv04]. High assurance systems are developed using a rigorous mathematical approach known as formal methods.

Former NPS students explored formal specification tools such as PVS [Ubh03] and Specware [Dec06] for their usefulness in formally verifying the Bell and LaPadula model and the TCX LPSK, respectively. These tools lack the ability to clearly specify state model semantics. The TCX framework aims to not only identify a high assurance toolset, but a toolset of standalone tools with ease of use and low learning curves. The successful dissemination and adoption of the framework by the growing high assurance development community will depend on not only the utility of the framework's toolset, but also in the ease of use of the toolset.

Alloy is a new form of model building based on the small scope hypothesis [Jac06]. The hypothesis states that if an inconsistency in a model exists there is a high probability that it will present itself within a small scope of the model. The ability to get feedback from the Alloy Analyzer in the form of model instances and counterexamples makes this tool distinct from older formal specification techniques. We set out to investigate if a relative novice to high assurance development in an "apprentice" like role can use Alloy to adequately describe the formal security policy model (FSPM) and the formal top level specification (FTLS) of a system like the TCX LPSK, prove their consistency, and demonstrate property preservation between their mappings.

Two documents exist for the TCX LPSK. One states the security policy of the TCX LPSK [Lev04] and the other describes the preliminary interface. The security policy document contains both the security policy and a FSPM written in predicate logic. The preliminary interface document contains a description of high level function names

with corresponding parameters and return values. We first produced an Alloy specification that matches the behavior stated in the security policy document.

The security policy consists of two predicates, one to restrict information flow and one (*TPO*) to restrict extraordinary flows, to *trusted subjects* (those that go beyond the identified partial ordering). We defined and tested the information flow predicate without complication. This was not the case for the trusted partial ordering predicate TPO. During testing the Alloy Analyzer produced models that did not conform to the trusted partial ordering. We ultimately determined the reason for the inconsistency was not a weakness of the Alloy specification language or of the security policy, but our particular expression of the intended security policy. Understanding this we constructed an alternative specification of the TPO which more clearly describes the intended security policy. This was a major success of our experimentation.

We then augmented the FSPM specification to create an FTLS. We did this by adding a predicate representing each class of interface in the preliminary LPSK interface document. Our models were helpful in demonstrating how the security properties hold between the security policy and the preliminary interface. Being that the interface is still in a preliminary stage our work was additionally useful to the designers in furthering its development.

We demonstrate that Alloy is an important tool to include in the toolbox of rapid high assurance development. Its low learning curve allows for the beginner to formal methods and high assurance systems to quickly hone their development skills. We also demonstrate that Alloy is useful even for more seasoned developers.

I. INTRODUCTION

A least privilege model for separation kernels (LPSK) is an example of a high assurance software system and a key component of a long term development project of the CISR group at the Naval Postgraduate School (NPS) known as the Trusted Computing Exemplar (TCX). An overarching goal of the TCX project is the establishment of an open framework for rapid high assurance development [Irv04]. High assurance systems are developed using a rigorous approach known as formal verification, which requires the use of mathematical, *formal methods*.

Formal methods tools, such as theorem provers and model checkers, help construct specifications for hardware and software and mathematically verify their correctness. Former NPS students explored formal specification tools such as PVS [Ubh03] and Specware [Dec06] for their usefulness in formally verifying security policies such as those in the TCX LPSK. These tools lack the ability to clearly specify state model semantics. Also, tool complexities prevented the students from fully implementing security policy models for the TCX LPSK.

The Alloy specification language provides a simple and clear language which should allow for modeling the TCX LPSK in its entirety. Work done by Jackson and others [Jac01, Has04] demonstrates Alloy's usefulness in memory protection and MLS models. Therefore, we set out to investigate if a relative novice to high assurance development in an apprentice-like role can use Alloy to adequately describe the formal security policy model (FSPM) and the formal top level specification (FTLS) models of an LPSK, prove their consistency, and the validity of the FSPM-FTLS mapping.

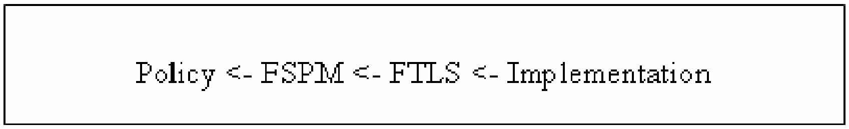
We present a basic overview of Alloy. Then we present our Alloy versions of the FSPM and the FTLS for an LPSK. Base on our experiences in developing the FSPM and the FTLS, we analyze the usefulness and limitations of Alloy in security system specification and proof. In this analysis we use the formal methods tool evaluation criteria developed by Ubhayakar and adapted by DeCloss. In addition to specifying the model in Alloy, we extended the formal specification of the TCX LPSK to include initial conditions and runtime state changes. The current developers of the TCX LPSK found this very useful.

II. BACKGROUND

A. FORMAL METHODS

The current industrial paradigm of software security involves constant patching of reused software. This paradigm is untenable. Security and safety critical software require an extremely high degree of assurance that they will behave as expected. Lives depend on the proper functioning of medical and aviation software. Lives depend on the proper containment of state secrets held on information systems. The confidentiality of large amounts of high value information (both personal and financial) in commercial-industrial databases needs better, high assurance protection. Formal methods are the preferred solution to the development of high assurance software.

The formal methods process rigorously verifies successive layers of software development in order to ensure that the system implementation enforces the security policy. At the top layer, a policy is conceived and then the policy is described in terms of a Formal Security Policy Model (FSPM). From there a Formal Top Level Specification (FTLS) is constructed, and then the final implementation [Bow03].



Policy <- FSPM <- FTLS <- Implementation

Figure 1. Formal Methods Mappings

The backward arrows in Figure 1 are used to indicate a mapping to the predecessor layer. This mapping usually involves rigorous assurance such as that provided by category theoretic morphisms. The effort of formal verification is eased if the specification language includes linguistic elements for describing the mapping between different levels of abstraction and can automatically generate theorems regarding the correctness of the mappings. However, the reader will notice that this is not possible between the Policy and FSPM layers, because the FSPM is intended to reflect the Policy’s English description. Consequently, assurance that the FSPM accurately reflects the Policy is accomplished by peer review.

Formal specification languages and related theorem provers, such as PVS, are common tools used in high assurance software development. Theorem provers are time consuming, costly, and limited in their ability to be fully automated. The assistance they require from a human operator is nontrivial. Model checkers, such as SPIN, use temporal logic to model the consistency of finite state machines in an automated fashion. Most software programs of any importance tend to be infinite state systems with properties that are computationally undecidable. Therefore, the success of model checkers has been mostly limited to hardware and they have not been well suited for the complexity of software. Current academic thought is that secure software requires a combination of formal languages and automated correctness proofs for correct development.

Alloy is a formal tool set that includes a specification language and a model analyzer. Alloy does not provide built-in syntactical elements for morphisms. However, it is well suited for incremental development. In our experimentation our init and runtime models correspond to the FTLS and are augmentations of the security model which corresponds to the FSPM. In the future work section we included a recommend approach for adding inter level mapping to Alloy models.

B. THE TCX

Over the years the computer security community has created many standards and documents detailing and outlining principles of computer security. Examples include the Trusted Computer System Evaluation Criteria (TCSEC), also known as the Orange Book, and the Federal Information Processing Standards (FIPS). The Common Criteria is an international standard that has attempted to build a unifying standard for the underlining requirements of a secure computer system. A computer system evaluated under the Common Criteria will receive an Evaluation Assurance Level (EAL) from 1 to 7, 7 being the best [Com05]. Formal methods are an integral part of the development of

EAL7 systems. The few commercial products which provide high assurance according to Common Criteria standards use proprietary methods, management, and code not open to the public.

The Trusted Computing Exemplar (TCX) is a project to develop an EAL7 system that is open to the public in the spirit of the open source movement. The TCX includes a formal specification of a least privilege separation kernel (LPSK) which will provide an open framework that other academic, commercial, and military institutions can corroborate and build on to provide high assurance software to the Navy, Department of Defense, and the national information infrastructure. The TCX framework aims to identify a set of interoperable standalone tools that feature ease of use and low learning curves [Irv04]. The successful dissemination and adoption of the framework by the growing high assurance development community will depend on not only the utility of the framework's toolset, but also in the ease of use of the toolset.

C. LPSK

The concept of a separation kernel was first proposed by Rushby in 1981[Rus81]. In recent years separation kernels have gained in popularity and are currently used in military avionics, military communications, and virtual machine monitors (VMM). A separation kernel partitions the system resources and controls the flow of information between the partitions. The separation kernel ensures that no information flows between partitions contrary to a set of allowed flows.

A separation kernel that allocates resources to partitions and processes in a fixed manner upon initialization is called a *static separation kernel*. For example, the kernel allots a fixed amount of processing time to each of its partitions. Dynamic time slices allow for the possibility of insecurities know as covert channels [Mil89]. Covert channel analysis can be very complex. Therefore, static separation kernels are desirable for simplicity of design, although at the expense of flexibility and efficiency of the system. The Principle of Least Privilege [Sal75] requires active resources in a system to not have more privileges than is absolutely necessary. Therefore a least privilege separation kernel (LPSK) is a separation kernel that also implements the Principle of Least Privilege.

In October 2004, the CISR group at the Naval Postgraduate School published *A Least Privilege Model for Static Separation Kernels* [Lev04]. It is a high-level design document for a LPSK and its security policy. The document contains both a description of the security policy and a FSPM written in predicate logic that specifies the critical elements of a LPSK, and a security predicate over all possible operations that could be included in a secure system. The initial and runtime conditions were not included in this model, but are operations contributed by our experimentation. The CISR group is currently in the process of producing an interface description for the TCX LPSK. This preliminary interface document describes the TCX LPSK's high level function names and corresponding parameters and return values. This provided the input for our FTLS of the TCX LPSK. Since the interface document was still in draft form, we worked with the authors to both ensure our understanding of the interface as well as to help ensure the consistency of the interface document.

III. OVERVIEW OF ALLOY

A. ALLOY’S INNOVATION

1. The Small Scope Hypothesis

Theorem provers are used to prove the properties of a specification (e.g., the policies of a system) with certainty. However, provers are complex tools, not fully automated, and very time consuming. Traditional model checkers have not been well suited for the complexity of software. However, the Alloy Analyzer is a new form of model building based on the “small scope hypothesis” that builds models from a semantic language of sets and first order predicate logic similar to Z.

The small scope hypothesis forms the basis for the use of model builders in contexts that require assurance of correctness. It states that if an inconsistency in a model exists there is a high probability that it will present itself within a small scope of the model [Jac06]. For example, in a model of a file system the number of files modeled is the “scope.” If no error or inconsistency is found by the model analyzer in a model of ten files, there is a very low probability that an error or inconsistency will present itself in a much larger model of a hundred files. This small scope hypothesis allows for the rapid verification of software specifications with much greater detail than other tools.

The small scope hypothesis is not without its skeptics. Formal methods in the strictest sense demand a certainty of correctness and do not allow for a probability of correctness. The Common Criteria’s highest level of security is EAL7. If a product obtains this level of evaluation it can be said that it was developed with tools that have sufficiently proven security. Nevertheless, in 2002, Clark Wiesman used Alloy to develop a specification of an assurance system with the goal of EAL7 evaluation [Wie02]. The Common Criteria only requires that formal tools be used in development, but makes no mention of the specific tools to be used, be they model checkers, theorem provers, or otherwise. “Where the functional specification is formal, the proof of correspondence between the TSP model and the functional specification shall be formal.” [Com05]

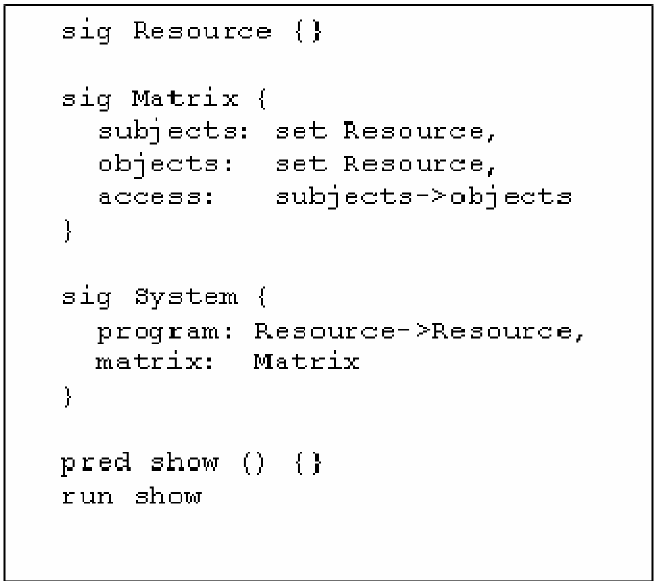
2. Immediate Visualization

Alloy connects the elegant world of specification languages with the power of the world of model checkers. Two great positive aspects of Alloy are the ease with which the user can express their model and the readiness with which the user can see if they have expressed what they intended. In essence, Alloy has automated a process that occurs with peer review. Alloy predecessors, such as Z, do not have this advantage. They require a large amount of effort to properly verify a model. Then more effort is required to verify with peers if the specification indeed models what is desired.

The Alloy tool set consists of the Alloy Analyzer and the Alloy specification language. The Alloy specification language has evolved from Z notation. The syntax defines sets and relations in such a way that it almost appears to be an object oriented programming language. The elegance and simplicity of this syntax allows for a large amount of natural expressivity. The Alloy Analyzer reads an Alloy specification, checks its syntax, and searches for specification inconsistencies within the defined scope. If the analyzer finishes, its list of inconsistencies, if any are present, is ensured to be complete. It then produces models that can be viewed as box or tree structured graphs. This provides immediate feedback that the user can use to understand what it is they have modeled.

The Alloy Analyzer first verifies the Alloy specification syntax and then converts it into a propositional logic formula – a normalized Boolean expression. The Alloy Analyzer then feeds the propositional formula into a third party satisfiability (SAT) solver which is used to generate either counter examples or models that can be immediately visualized. The ability to get feedback from the Alloy Analyzer in the form of model instances and counterexamples makes this tool distinct from older formal specification techniques.

Figure 2 shows a simple Alloy specification of system containing an access matrix. Figure 3 shows a graph visualization of a model that conforms to the constraints of the Figure 2 specification. One thing that we see immediately is multiple "systems" in the model. This is because the specification did not specify only one system should exist.



```
sig Resource {}

sig Matrix {
  subjects:  set Resource,
  objects:   set Resource,
  access:    subjects->objects
}

sig System {
  program: Resource->Resource,
  matrix:  Matrix
}

pred show () {}
run show
```

Figure 2. A Simple Alloy Specification

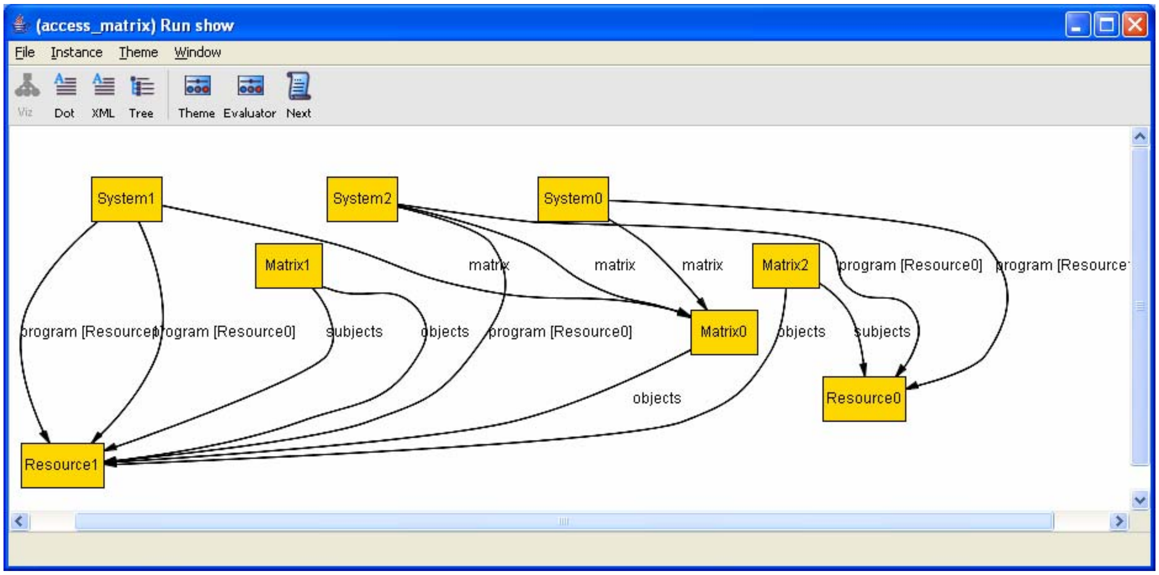


Figure 3. Visualization Corresponding to Simple Alloy Specification

3. SAT Solvers

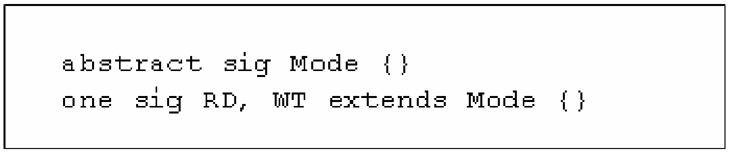
The Boolean satisfiability problem consists of assigning values to a normalized Boolean expression (and’s and or’s only) such that it results in a true expression. In complexity theory the problem is considered non-deterministic polynomial time complete (NP-complete). This means that a fool-proof fast solution to the problem is unlikely to ever be found. However, SAT solvers use heuristics such as pruning to quickly find solutions in a search space (i.e., within the defined scope). Many good SAT solvers are freely available and new improved SAT solvers are frequently released.

There may be many assignments of values that result in a true expression. Therefore, there may be multiple solutions. If no such assignment is possible within the defined scope then there is no solution. The Alloy Analyzer offers a variety of SAT solvers. The SAT4J solver will produce multiple solutions when multiple solutions exist. With the SAT4J solver the user can click the next button in the visual display area to view other satisfying models. If no satisfying model is found, Alloy will produce an error message that the model may be inconsistent.

B. ALLOY’S LANGUAGE

1. Syntax

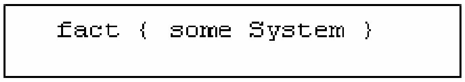
The ‘*sig*’ keyword (short for signature) is used to declare sets. The declared sets can be operated on much as if they were objects in an object oriented programming language. The ‘*extends*’ keyword will create disjoint subsets. The '*abstract*' keyword indicates that an item should not be created in the model, but that items extended from it may. The '*extends*' and '*abstract*' keywords are analogous to object oriented programming with a hierarchy of parent and child classes. In Figure 4 the '*one*' keyword is a multiplicity keyword that means one and only one item should be in the model.



```
abstract sig Mode {}
one sig RD, WT extends Mode {}
```

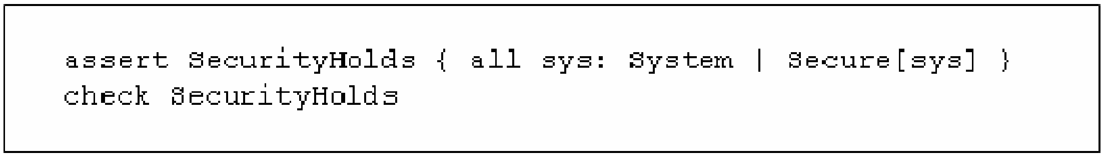
Figure 4. abstract and extends Keywors

The ‘*pred*’ keyword declares a predicate that returns only a true or false value. The ‘*fun*’ keyword is used for functions and both functions and predicates may take parameters. The ‘*fact*’ keyword can be used anywhere in the specification to place constraints on the model. The example in Figure 5 would require at least one System item be in the model. That is, it would eliminate any model with an empty System as a model that satisfies the specification. The ‘*assert*’ keyword is use to verify that certain conditions hold in the specification. The example in Figure 6 would verify that every System in the model conforms to the Secure predicate. If this is not the case the Alloy Analyzer would produce a counter example.



```
fact { some System }
```

Figure 5. Example of a Specification Constraint

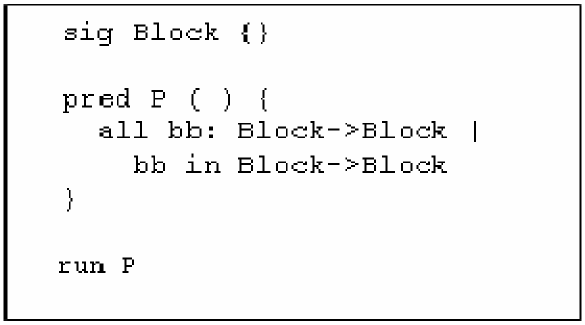


```
assert SecurityHolds { all sys: System | Secure[sys] }
check SecurityHolds
```

Figure 6. Example of Model Assertion

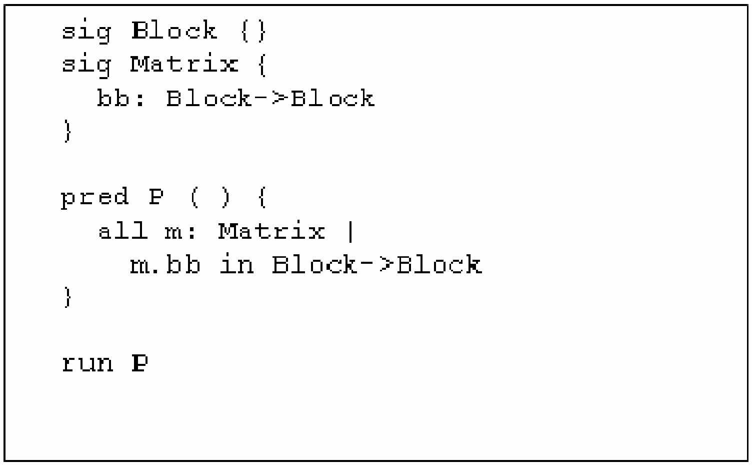
2. A Mix of Relational and Predicate Logic

Alloy uses an elegant mix of relational logic and first order predicate logic. It is not higher ordered. This means that quantification can not directly occur over functions, relations, or predicates. However, this doesn’t appear to be problematic and there is almost always a way to restate such a problem. (See page 41 of Software Abstractions) This did take some time to become accustomed to. The specification in Figure 7 states the obvious, that is, every possible block to block relation is in the set of block to block relations. Figure 7 will give a higher order compiler error. However, we were able to work around this by wrapping the relation into a signature as shown in Figure 8.



```
sig Block {}

pred P ( ) {
  all bb: Block->Block |
    bb in Block->Block
}

run P
```

Figure 7. Quantification Over A Relation

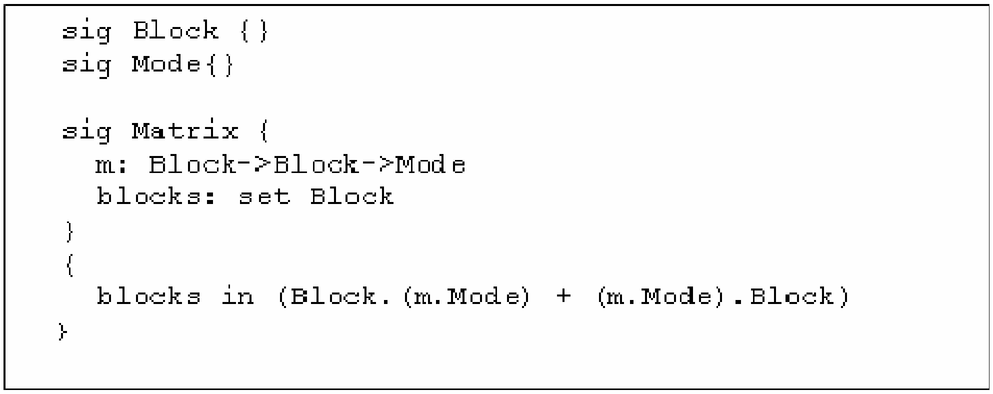


```
sig Block {}
sig Matrix {
  bb: Block->Block
}

pred P ( ) {
  all m: Matrix |
    m.bb in Block->Block
}

run P
```

Figure 8. Relation Wrapped In A Signature

3. The Join Operation

Most students will recall the join operation from their set theory or discrete math classes. The join operation is what provides most of the power and simplicity to Alloy. With a brief refresher the student will be prepared to construct models in Alloy. In a join operation, the last item of one tuple is matched the first item of another. All the items except the matching tuple items are returned. (See page 57 of Software Abstractions) The dot join operation can be used in Alloy to access members of a signature, implying the relational nature of the signatures members with the signature itself. The dot join is frequently used in Alloy with a set and a relation. This is done to find the members of one part of relation such as in Figure 9. In this example the set Mode is joined with the relation m to produce just the Block->Block part of the relation m. Then set Block is joined with the resultant value to produce a set of Blocks from the relation.



```
sig Block {}
sig Mode{}

sig Matrix {
  m: Block->Block->Mode
  blocks: set Block
}
{
  blocks in (Block.(m.Mode) + (m.Mode).Block)
}
```

Figure 9. Join Used With Relations And Sets

4. The Transitive Closure

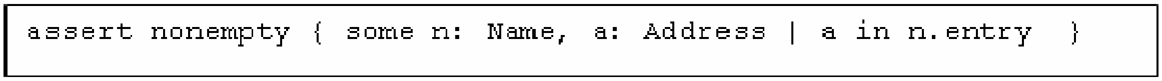
In relational logic we often take the transitive closure of a set with respect to a relation in the set. Usually the transitive closure is constructed using either iteration or recursion. Neither Alloy nor predicate logic possesses iteration or recursion. There may be specific models where one can simulate the transitive closure, but in the general case the transitive closure is impossible to construct. (See page 234 of Software Abstractions) Consequently, Alloy has a built in operation to construct the transitive closure. The operation is represented by the ^ (carrot) symbol. The operation's inclusion into to the Alloy specification is of tremendous significance. We use this at key points in our model and it is possible that our model may not have been possible without it.

5. Libraries

The Alloy developers have adopted the design philosophy of keeping the core syntax simple and while allowing the language to grow and be enriched by robust libraries. In our experimentation we used a standard utility library for common functions and a library for constructing orderings and sequences. In the future works section we propose library development for demonstrating morphisms.

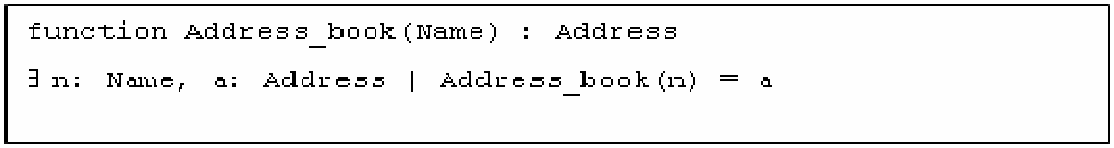
C. LESSONS LEARNED

The average student with knowledge of predicate and relation logic should be able to pick up Alloy with minimal effort. However there are occasional subtleties of predicate logic that the average student could quickly stumble over. One such example [Jac06, p.71] that caused us some effort is shown in Figure 10. This states that the Address book mapping names to address is non empty. If the empty address book is not eliminated from the model, the Alloy Analyzer will produce the empty address book as a counter example. Figure 11 shows how this example might look in predicate logic.



```
assert nonempty { some n: Name, a: Address | a in n.entry  }
```

Figure 10. Example From Page 71



```
function Address_book(Name) : Address
∃ n: Name, a: Address | Address_book(n) = a
```

Figure 11. Example In Predicate Logic

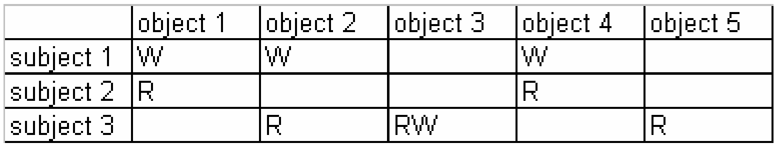
As with any new language, the beginning student would be wise to always ignore default values and explicitly state the desire value. In particular the default multiplicity in Alloy is one. For example, sig Matrix { blocks: Block }, by default means that blocks contains exactly one Block, when what is really desired is a set of blocks. We could have saved several hours by always stating explicitly one, set, lone, etc, in this case sig Matrix { blocks: set Block }.

Without a doubt advances in SAT solvers and hardware have greatly facilitated the connection between specification languages and model analyzers. While working on this thesis Alloy was upgraded from version 3 to 4, with a noticeable increase in performance. It appears that hardware and heuristics are continuing to get better and so the future for Alloy and the small scope hypothesis is bright.

IV. SECURITY MODEL

A. SECURITY FRAMEWORK

Butler Lampson was the first to propose the concept of an access matrix to analyze the information interaction of a system [Lam71]. The access matrix is a representation of a system which abstracts system resources into active entities called subjects and passive entities called objects. A resource is anything in the system we wish to consider, be it a file, a process, a user, a program, firmware, etc. The objects are usually placed along the columns of the matrix and the subjects along the rows. The cells of the matrix represent modes of access, usually a read, write, execute, or some combination there of. The matrix can then be used to study the flow of information in the system. For example, in Figure 12 subject 1 can write to object 1 and subject 2 can read from object 1 therefore information can flow freely from subject 1 to subject 2.



	object 1	object 2	object 3	object 4	object 5
subject 1	W	W		W	
subject 2	R			R	
subject 3		R	RW		R

Figure 12. Access Matrix

The Bell and LaPadula model (named for its authors) is a confidentiality policy. The model places partially-ordered security labels on system resources [Bel73]. The model allows subjects to write up and read down the ordering, but prohibits them from writing down or reading up. This policy increases the utility of the system by allowing multiple sensitivity levels to operate in the system, while at the same time protecting the system from Trojan programs that try to surreptitiously downgrade the sensitive information.

The Principle of Least Privilege was first proposed by Saltzer and Schroeder [Sal75]. As its name suggests, it requires system resources to only have access privileges that are absolutely necessary. This reduces potential vulnerabilities by eliminating unnecessary functionality. Also, the elimination of discretionary information flows eases the effort in analyzing information flow. All high assurance systems should include the Principle of Least Privilege in their design to reduce flaws and accidental error. Applied to the Bell and LaPadula model, the Principle of Least Privilege could prohibit a low resource from writing up if the low resource has no compelling reason (e.g., per the design) to do so.

A separation kernel is devoid of any intrinsic security policy. It only partitions and isolates system resources per its configuration data and allows specified flows between the blocks of the partition. An application on top of the separation kernel can then transpose a multilevel security policy on the flow of information between the partitions by associating labels (top secret, secret, etc) on the individual blocks. Applications can also restrict (e.g., transitive) flows to establish their own policies as subsets of the flows allowed by the separation kernel. This allows for separating the functionality of isolation and policy interpretation, reducing the complexity of analyzing the kernel and may make the task of evaluating a separation kernel more manageable than a kernel in which the MLS labels are embedded.

Unfortunately, while separation kernels have gained in popularity, many lack the ability to enforce the Principle of Least Privilege. The paper *A Least Privilege Model for Static Separation Kernels* [Lev04] has attempted to address this issue. It details a Formal Security Policy Model (FSPM) for such a kernel. In our experimentation we followed the FSPM as outlined in the paper to develop an Alloy specification. The model we developed contains two essential access matrices that are intended to be orthogonal to each other. One represents the flow of information between partitions in the kernel and, as required by the “TPO” predicate to be discussed later, should conform to the Bell and LaPadula model. The other represents the Principle of Least Privilege. Once the kernel is up and operational it will remain static. This means that the flows of information as allowed by the two matrices will not change during runtime.

B. SPECIFICATION IN ALLOY

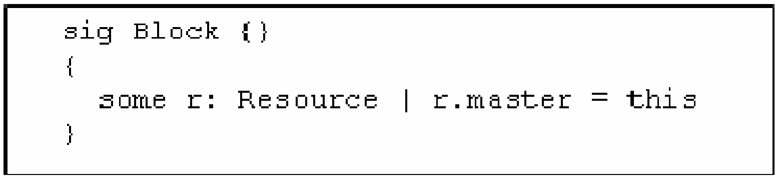
1. Resources, Modes, and Blocks

We produced an Alloy specification that matches the FSPM as stated in section 5 of the paper, *A Least Privilege Model for Static Separation Kernels* [Lev04]. The Alloy specification language is well suited for a gradual, incremental development style. This allowed us a quick start at specifying the basic elements of the TCX LPSK. Once we defined resources, subjects, access modes, and blocks we used the Alloy Analyzer to visualize numerous representative models. Initially the Alloy Analyzer produced trivial and simplistic representative models. However, without much difficulty we added minor constraints in order to view more interesting models.

We used identifiers similar to those in the paper. We used the Operation construct in our initial experimentation and included it in our definition of the Secure predicate. However, we noticed that it added little to the description of the model. In order to understand other aspects of the model we simplified the model by removing the Operation construct entirely. For this reason it is absent in our final model.

The word subject and object are typically used to indicate which resource is the active entity performing the read and/or write on another passive resource. In our model we use the subject terminology and we allow one Subject to operate on another subject (i.e.,. one process communicating to another process). We drop the object nomenclature and simply refer to a subject accessing a resource (which may be another Subject).

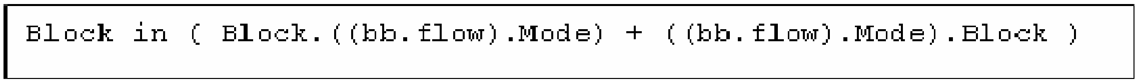
In the specification every Resource element is assigned exactly one and only one Block element. Thus the Block elements partition the Resource elements into equivalence classes as is intended. The separation kernel is static, therefore there is no reassigning of Resource elements to a different Block once the kernel is running and it will not make sense to have an empty Block. The Figure 13 achieves this constraint.



```
sig Block {}
{
  some r: Resource | r.master = this
}
```

Figure 13. No Empty Blocks

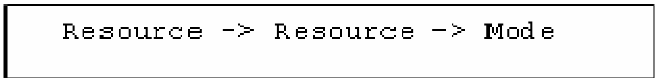
Of course we are only interested in resources and blocks that are inside of our System. Because all Resource elements belong to a Block, Alloy will not produce Resource elements or Block elements that have nothing to do with each other. In like manner we need to instruct Alloy that every Block participates in the System to prevent Alloy from producing models with Resource elements and Block elements that are outside of the System. In the System signature we add this constraint as seen in Figure 14.



```
Block in ( Block.((bb.flow).Mode) + ((bb.flow).Mode).Block )
```

Figure 14. All Blocks In The Relation

2. MM, SR, and BB



```
Resource -> Resource -> Mode
```

Figure 15. Access Matrix In Alloy

In Alloy, a 3-tuple or 3 item relation can be used to indicate an access matrix or set of flows as in Figure 15. Our model of the system will contain two access matrices. One represents the flow of information between blocks (BB) and the other the flow of information between resources (SR). The two matrices are intended to be orthogonal, that is, there may be flows allowed in one that are not allowed in the other. These matrices only determine what flows are allowed. The MM matrix represents the flows that are actually realized by the system in operation. MM should be orthogonal to SR and BB. That is SR/BB is the policy of what is allowed, and MM is what the program wants to do. A secure system is a system in which all flows in MM are allowed by both access matrices BB and SR. Conversely, an insecure system is a system that contains at least one flow in MM not allowed by both BB and SR. The security predicate states that a secure system allows only the program actions (MM) that conform to the policy (SR/BB).

A Block element may contain any combination of Subject elements and Resource elements. Therefore, the Block nomenclature is devoid of the Subject and Resource distinction. In order to better visualize and talk about the flow of information only Subjects are allowed to read and/or write. That is, Resource elements that are not also Subject elements should never initiate a flow. Therefore, our SR is the relation Subject->Resource->Mode. Technically SR is not required to be a subset of BB. That is, it may allow flows that are prohibited by BB, however, in a secure system those flows would never be realized, because what is allowed is only the intersection of the two matrices.

In the system signature of our Alloy specification, MM and sr_flow represent MM and SR respectively. However, we will be interested in taking the partial ordering and the *trusted partial ordering* of the block flow matrix BB. To help with the trusted partial ordering we created a specific structure for BB. Instead of creating a separate predicate for FLOWS as is recommend in the paper we added the concept directly to BB. We can consider a read originating from Block A to Block B to be the same as a write originating from Block B to Block A in terms of information flow. We can simplify either statement by saying information has flowed from Block B to Block A. In our BB signature we call this the basic_flow and use FLOWS for the transitive closure of the basic flows.

From a security perspective the transitive closure is useful for seeing all blocks that information can eventually flow into. However, as we mentioned above, in the general case the transitive closure is not possible in predicate logic. Our solution for both the BB access matrix and the trusted partial ordering predicate TPO involves the transitive closure (^ carrot symbol in Alloy). It may be possible to re-factor both and eliminate the need for transitive closure. Time constraints prevented us from pursuing this further.

3. PO

We are interested in a traditional confidentiality policy between the blocks that prohibits reading up or writing down. The separation kernel itself only isolates the blocks and regulates the flow between them, but is devoid of semantics labeling one block greater than another.

However, we can define a predicate that will require the flows between blocks to be defined in such a way that information is not allowed to flow circularly. That is if information leaves a block, there is no transitive flow that will lead back to itself. It is important to note that any two blocks are not required to be related by a flow. For example, BlockA->BlockC, BlockB->BlockC, but there is no flow between BlockA and BlockB and no indication if one is greater than the other. Therefore such a non-circular flow of information is technically considered a partial ordering but not a total ordering. That is, not all the items in the set are comparable.

Mathematically a partial ordering (PO) is defined as a relation over a set that is transitive and antisymmetric. Antisymmetric means there is no circularity in the relation. Many authors extend a partial ordering to include reflexivity which means every item in the set is comparable to itself. We include reflexivity in PO allowing it to be commented out if not desired. The definition in the paper and our corresponding specification in Alloy are both straight forward and derived without complication.

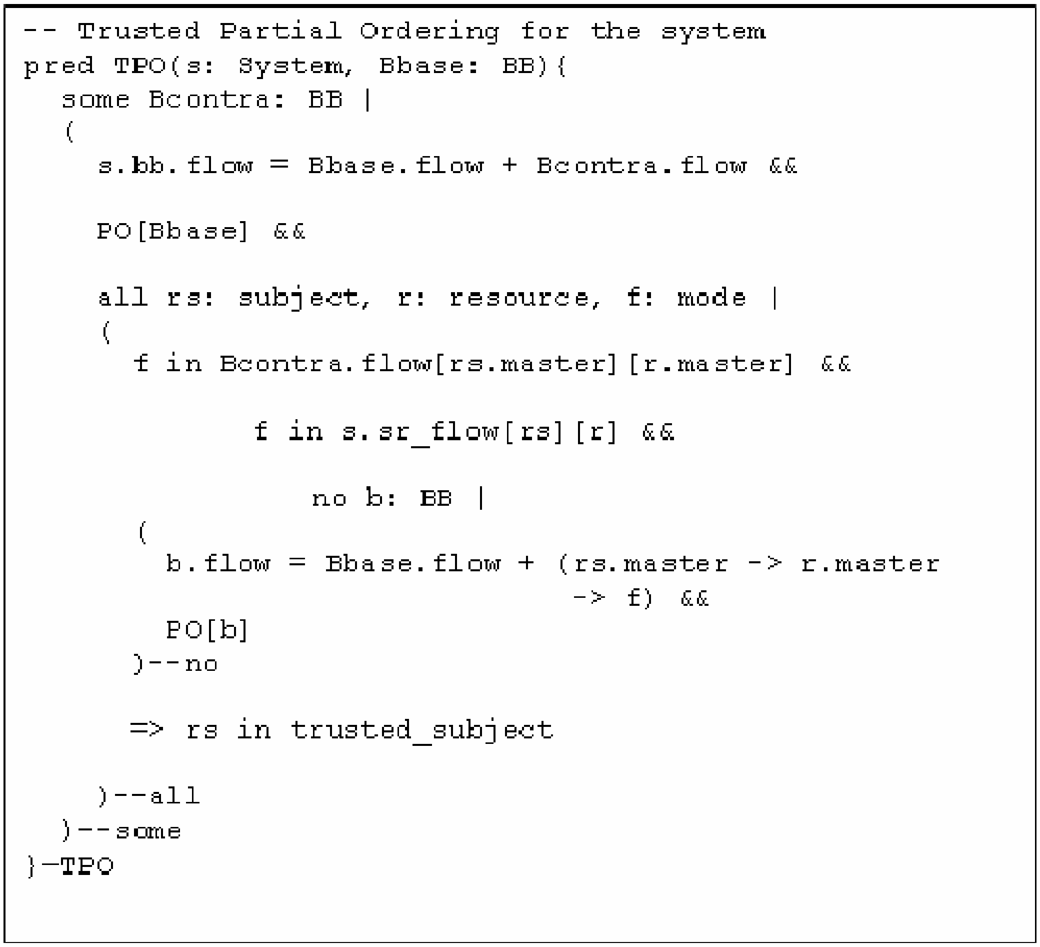
It should be noted that one often hears people speak of lattices in information security. A lattice is a partial ordering where every comparable pair of items contain both a least upper bound (lub) and a greatest lower bound (glb), which results in a universal upper and lower bound to the lattice.

4. TPO

Overtime an item of information may no longer be sensitive and its sensitivity may need to be downgraded. To do so requires violating the partial ordering. Therefore the notion of a trusted subject is introduced. A trusted subject is a subject (i.e., process) that has undergone rigorous analysis and is trusted not to downgrade information other than the information it is intended to downgrade.

A trusted subject is allowed to violate the partial ordering but it is not required to violate the partial ordering. Therefore, there may be flows in the system that are the result of the trusted subjects and violate the partial ordering. However, not every trusted subject necessarily violates the partial ordering. The challenge here is that we do not really have a way of identifying which subject, trusted or otherwise, caused a flow in bb, after all, the matrices are intended to be orthogonal. However, the reverse may be possible. We can identify the flows in sr_flow and require any flow that upsets the partial ordering in bb to be a trusted subject. However, this also requires us to identify the partial ordering.

Figure 16 shows our initial experimentation where we implement the solution exactly as it is stated in section 5 of the paper *A Least Privilege Model for Static Separation Kernels* [Lev04]. This was both a major source of effort and a major success of our experiment. During testing the Alloy Analyzer produced models that did not conform to a trusted partial ordering. We initially assume that we had incorrectly specified the TPO in Alloy or made some other flaw in methodology. We spent considerable effort in re-specifying the predicate with the block-flows Bbase and Bcontra stated in alternative ways. After exhausting all these possibilities, we ultimately determined the reason for the inconsistency was not a weakness of the Alloy specification language nor of the security policy, but in the ability of predicate logic to clearly describe the intended security policy.

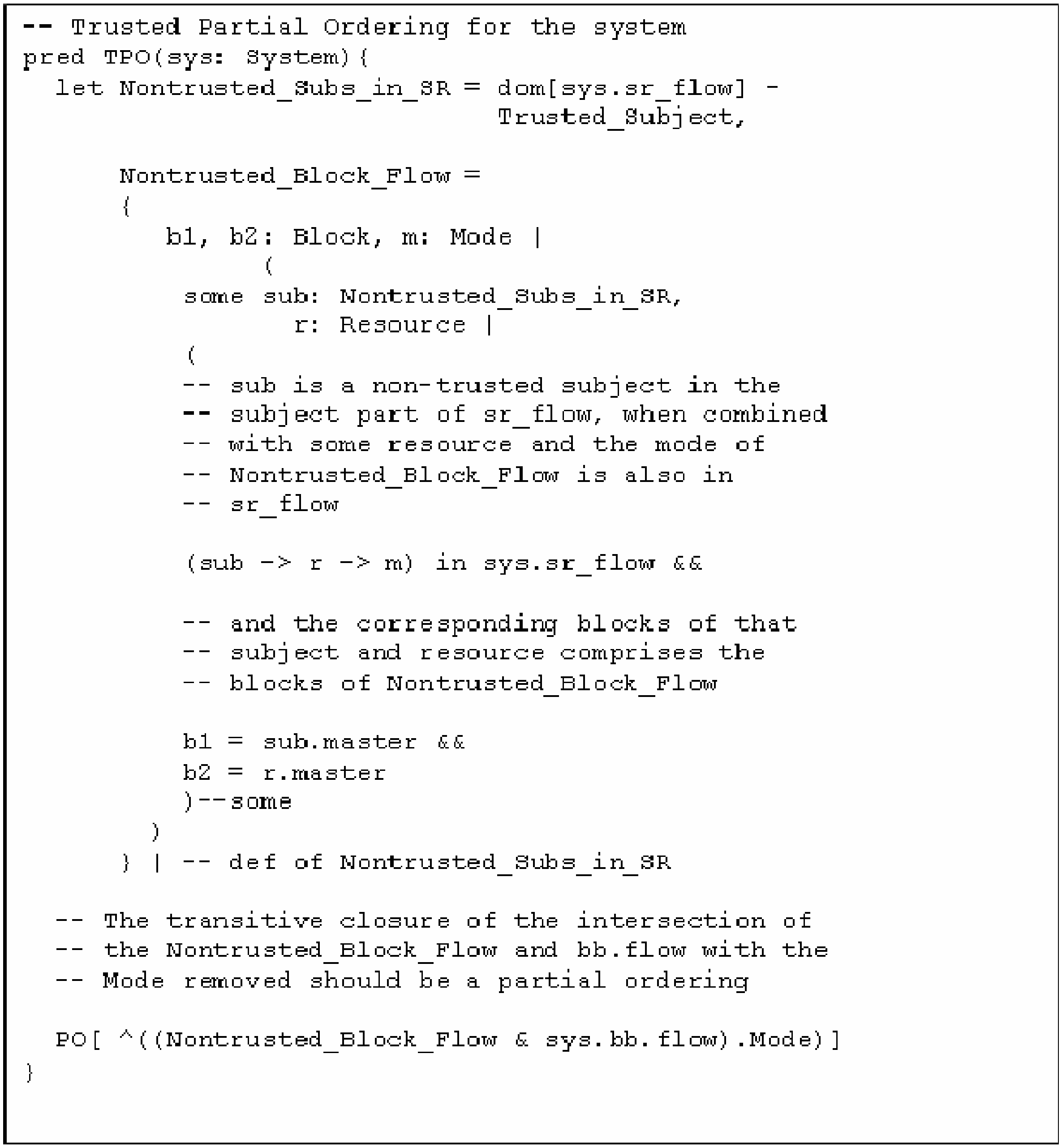


```
-- Trusted Partial Ordering for the system
pred TPO(s: System, Bbase: BB){
  some Bcontra: BB |
  (
    s.bb.flow = Bbase.flow + Bcontra.flow &&

    PO[Bbase] &&

    all rs: subject, r: resource, f: mode |
    (
      f in Bcontra.flow[rs.master][r.master] &&

             f in s.sr_flow[rs][r] &&

                no b: BB |
      (
        b.flow = Bbase.flow + (rs.master -> r.master
                                    -> f) &&
        PO[b]
      )--no

      => rs in trusted_subject

    )--all
  )--some
}—TPO
```

Figure 16. TPO As State In The Paper

When we specified the original documentation’s TPO predicate using the Alloy specification language the Alloy Analyzer produced the following counter example: Two block-flows that individually do not upset the partial ordering, but when taken together do upset the partial ordering. Understanding this we constructed an alternative specification of the TPO which more clearly describes the intended security policy.

This was a major success of our experimentation. Previous students who used tools such as PVS and Specware were not able to attempt modeling the TPO due to tool complexities. Consequently this issue went undiscovered. The major difference in our solution’s approach, as seen in Figure 17, from that of the paper is that our approach tests the entire set of non-trusted flows while the paper attempts to identify each trusted flow one at a time. Unfortunately this results in the partial ordering being identified via the SR least privilege matrix. This may have the same effective result, but weakens the concept of SR and BB being entirely orthogonal and makes visualizing the trusted partial ordering via BB difficult. As a precaution we included the transitive closure in our solution.



```
-- Trusted Partial Ordering for the system
pred TPO(sys: System){
  let Nontrusted_Subs_in_SR = dom[sys.sr_flow] -
                              Trusted_Subject,

      Nontrusted_Block_Flow =
      {
         b1, b2: Block, m: Mode |
               (
          some sub: Nontrusted_Subs_in_SR,
                 r: Resource |
          (
          -- sub is a non-trusted subject in the
          -- subject part of sr_flow, when combined
          -- with some resource and the mode of
          -- Nontrusted_Block_Flow is also in
          -- sr_flow

          (sub -> r -> m) in sys.sr_flow &&

          -- and the corresponding blocks of that
          -- subject and resource comprises the
          -- blocks of Nontrusted_Block_Flow

          b1 = sub.master &&
          b2 = r.master
          )--some
        )
      } | -- def of Nontrusted_Subs_in_SR

  -- The transitive closure of the intersection of
  -- the Nontrusted_Block_Flow and bb.flow with the
  -- Mode removed should be a partial ordering

  PO[ ^((Nontrusted_Block_Flow & sys.bb.flow).Mode)]
}
```

Figure 17. Our TPO

C. CONCLUSION

The security of the system is defined in terms of as information flows constrained by two access matrices, one of which conforms to a trusted partial ordering. Except for minor issues, specifying the FSPM in Alloy went smoothly. The major stumbling block was not the Alloy environment, but the formal definition of a trusted partial ordering. The major effort in overcoming this was not grappling with the Alloy specification or the Alloy Analyzer, but in grappling with our assumption of the formal definition of the TPO. The fact this issue has gone unnoticed before our experimentation has demonstrated that Alloy is not only useful for the novice, but for the more seasoned formal methods developer as well.

V. EXTENDED MODELS

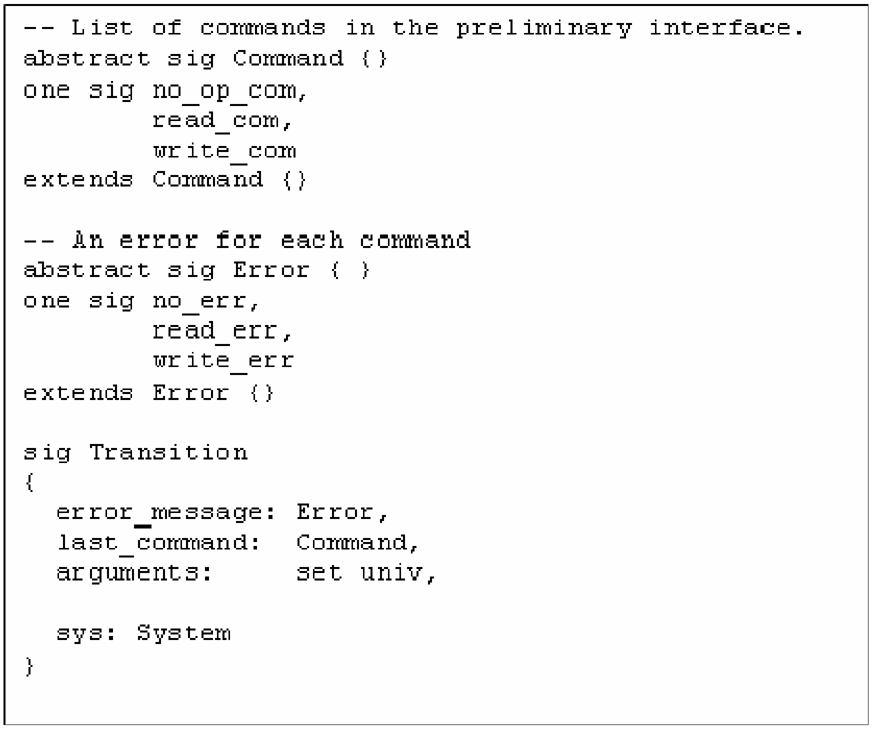
A. SECURITY FRAMEWORK

In the formal methods process the formal top level specification (FTLS) is a high level design specification that represents a refinement of the abstract concepts and properties of the policy model (FSPM) in the form of a general blue print towards building the actual system. The goal of the formal methods development process is to prove that the FTLS accurately represents the system in its entirety while preserving the properties of the FSPM. The separation kernel model that we built in section IV represents the FSPM of the separation kernel. Alloy allows for a gradual incremental development style. We can reuse the separation kernel Alloy specification unaltered in a new augmented Alloy specification. This augmented specification corresponds to a FTLS. Since it is nothing more than an augmentation of the FSPM specification we can declare that it represents the FSPM in its entirety.

Demonstrating that the FSPM properties have been preserved in the FTLS presents an interesting problem. We are interested in discovering insecurities in the design and implementation of the operating system. A traditional way to formally represent the security of a system is with a state transition model, where inputs on a state define a transition to a new state. By defining a set of possible inputs and an initial state we can model all reachable states. Additionally, by defining what makes a state valid or invalid we can check if any of the reachable states are invalid states, as well as identify invalid state-to-state transitions.

In our FSPM model our System signature and matching Secure predicate naturally correspond to states and their definition of validity. An informal system interface document defines the operations that can occur in the system. We use these interface operations for our set of inputs. By declaring our initial system secure, we check to see if any of the interface operations (inputs) lead to an insecure system. If all inputs result in secure states, the system is said to be “Secure”. This is often referred to as the basic security theorem.

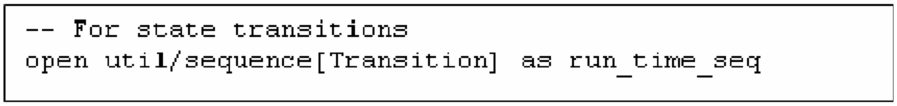
We created a Command signature to characterize the operations and related parameters and specified a subset of the actual operations. All the system operations have the ability to return error codes, as do the corresponding commands. In our initial experimentation we used the System signature alone to represent a state. This made it difficult to take advantage of the visualization ability of Alloy. Therefore, we created a Transition signature to encapsulate the state and the inputs as seen in Figure 18. Alloy has a built-in type to represent the universal set. This permits any type in the model to belong to the set, and enables our expression of state transitions.



```
-- List of commands in the preliminary interface.
abstract sig Command {}
one sig no_op_com,
        read_com,
        write_com
extends Command {}

-- An error for each command
abstract sig Error { }
one sig no_err,
        read_err,
        write_err
extends Error {}

sig Transition
{
  error_message: Error,
  last_command:  Command,
  arguments:     set univ,

  sys: System
}
```

Figure 18. Transition Signature In runtime.als

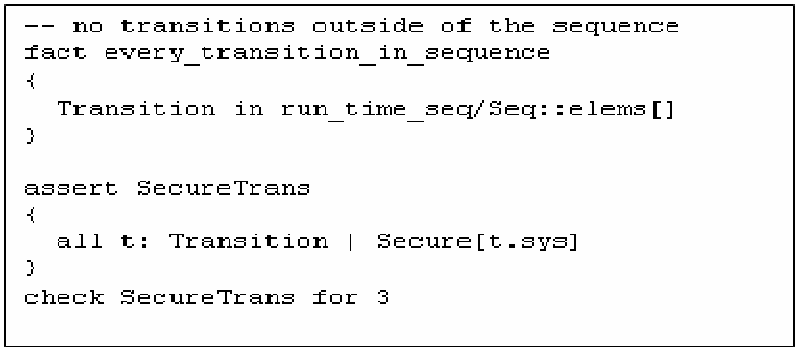
We encountered two issues when attempting to model Transition sequences. In our original experimentation we used the default ordering module. The ordering module constrains every element of the set being ordered to exist in the ordering. Consequently only one ordering is allowed. One potential difficulty the ordering module may have presented would have been in connecting the last state of the initialization module with the first state of the runtime module. After several trials experimenting with the ordering module we encountered issues due to its subtleties. Consequently we abandoned it and chose to use the sequence module in its stead as seen in Figure 19.



```
-- For state transitions
open util/sequence[Transition] as run_time_seq
```

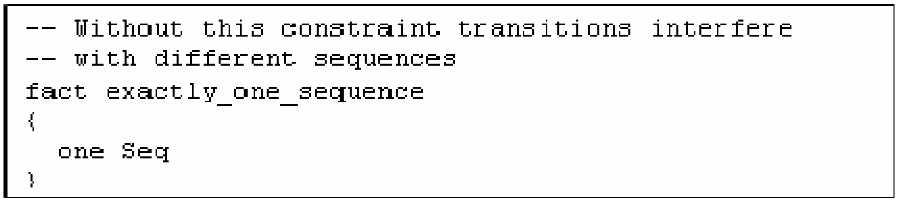
Figure 19. Declaration Of Sequence In runtime.als

The sequence module had its own subtleties that cost us a noticeable amount of time to resolve. Unlike the ordering module, elements may exist in the model that have no corresponding index to a sequence. Also, their may be more than one sequence to a model. To ensure that state transitions extraneous to a sequence did not interfere with the model, it seemed reasonable to simply declare that every state transition was included in the sequence and then check to see if every reachable state was secure as seen in Figure 20. However, when multiple sequences were created in the model a state transition in one of the sequences would interfere with another sequence creating inconsistencies in the model. For this reason we limit the number of sequences to exactly one as seen in Figure 21.



```
-- no transitions outside of the sequence
fact every_transition_in_sequence
{
  Transition in run_time_seq/Seq::elems[]
}

assert SecureTrans
{
  all t: Transition | Secure[t.sys]
}
check SecureTrans for 3
```

Figure 20. All Transitions In The Sequence



```
-- Without this constraint transitions interfere
-- with different sequences
fact exactly_one_sequence
{
  one Seq
}
```

Figure 21. Only One Sequence In The Model

B. TWO SEPARATE MODELS

Our interface design document has essentially broken the system down into two distinct but connected sub-systems. One that represents the system at the point it is initialized and another that represents system during runtime, in a fully functional capacity. Consequently we developed two distinct Alloy specifications to represent the FTLS as distinct sub-systems. The connectedness of the two systems can be demonstrated by having the last state of the initialization FTLS be the first state of the runtime FTLS, as seen in Figure 22. Both incorporate the FSPM in its entirety, so the connection of each FTLS with the FSPM is implicit.

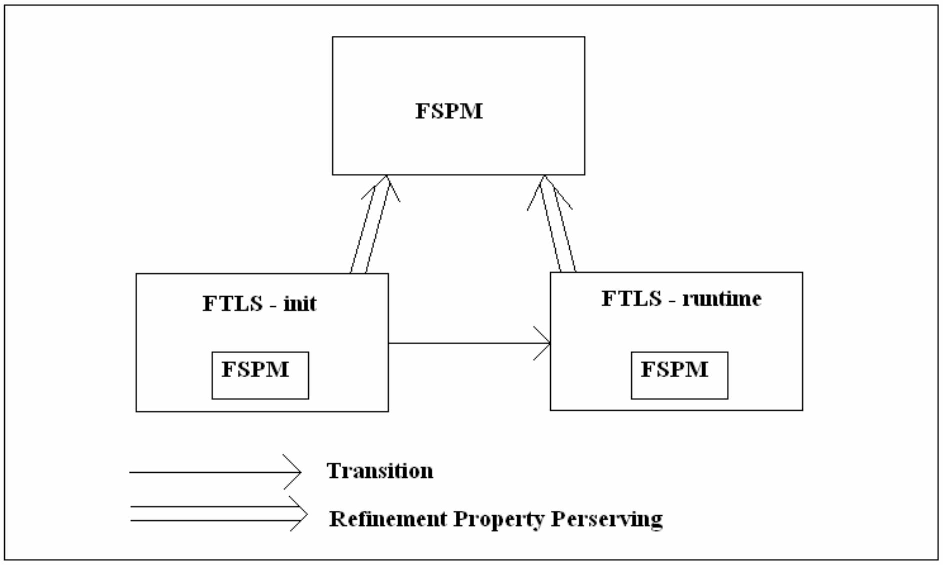
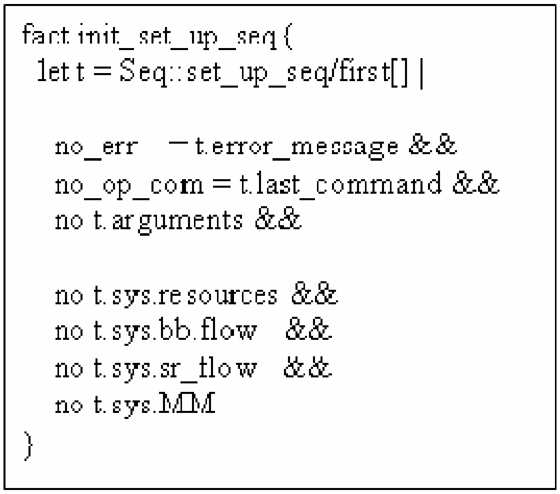


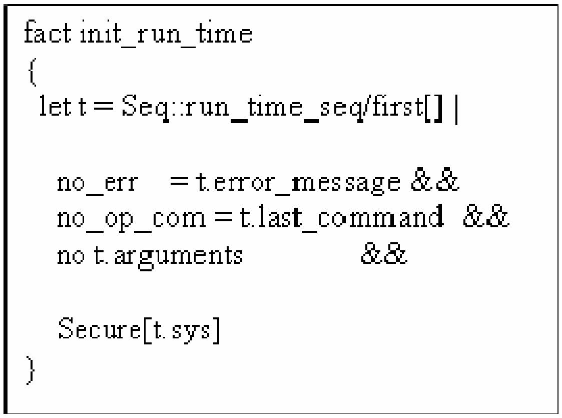
Figure 22. Relationship Of Models

To facilitate the connection between the two we added unused constructs in one model to the other. The major difference between the two models is the final two facts. The first fact declares the initial state of model. Figure 23 shows the specification fact for declaring the first state of the init model consisting of empty matrices and Figure 24 shows the specification fact for declaring first state of the runtime model consisting of established matrices. The second fact in each model defines a transition in the system as any of the allowed operations occurring. We then write separate predicates for each operation which allows for either success or failure of the operation. Alloy has a construct for *if then else* which we used in the transition fact to articulate the system response whether or not the operation was successful.



```
fact init_set_up_seq {
 let t = Seq::set_up_seq/first[] |

   no_err   = t.error_message &&
   no_op_com = t.last_command &&
   no t.arguments &&

   no t.sys.resources  &&
   no t.sys.bb.flow   &&
   no t.sys.sr_flow   &&
   no t.sys.MM
}
```

Figure 23. Init Fact



```
fact init_run_time
{
 let t = Seq::run_time_seq/first[] |

   no_err   = t.error_message &&
   no_op_com = t.last_command  &&
   no t.arguments            &&

   Secure[t.sys]
}
```

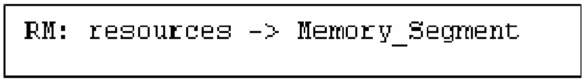
Figure 24. Runtime Fact

We are modeling a static separation kernel. This means that the allowed information flows remain unchanged while the system is up and operational. However, while the system is being initialized the allowed information flows are configured for the first time. Therefore, in the runtime model none of the matrices are allowed to change. However, in the initialization model we start with empty matrices and fill the matrices in until the initialization phase completes.

1. The Runtime Model

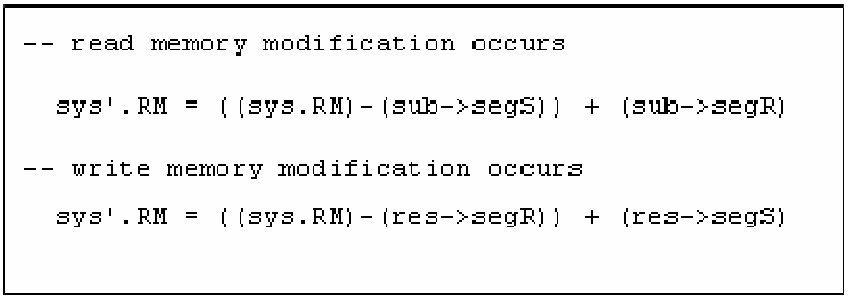
The interface document has various operations all dealing essentially with memory segments. From a security perspective what we are ultimately concerned with is the flow of memory. The most basic operation in the interface is reading or writing a byte of memory. Being a static kernel, resource allocation does not occur during run-time. Therefore, we only modeled a read and a write operation. In order to simplify the modeling of reading and writing we declared a signature Memory_Segment independent of the Resource signature. We left processes and resource handles out of this model. Figure 25 shows a relation we added to the System signature that maps every resource in the system to a memory segment. A successful read or write will update this relation accordingly. Extensions to deal with non-memory objects are left for future work.

Only subjects are allowed to initiate a memory flow in our model. When a subject reads a byte of memory from another resource it is overwriting one of its own bytes of memory with one of the resources bytes of memory. Likewise, when a subject writes a byte of memory it is overwriting one of the resource's bytes of memory with one of its own. By removing and adding subject and resource memory segments to the resource memory segment relation in the system signature appropriately we can model the flow of information exactly. In Figure 26 sys represents the current state system and sys' the state of the system after modification.



```
RM: resources -> Memory_Segment
```

Figure 25. Added To The System Signature



```
-- read memory modification occurs

  sys'.RM = ((sys.RM)-(sub->segS)) + (sub->segR)

-- write memory modification occurs

  sys'.RM = ((sys.RM)-(res->segR)) + (res->segS)
```

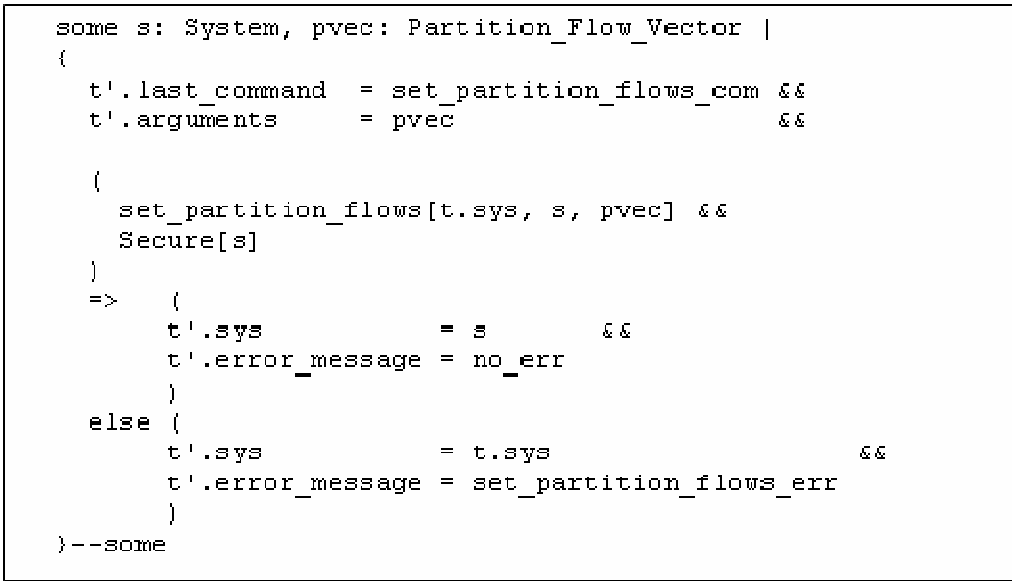
Figure 26. Read And Write Defined

2. The Initialization Model

In our first fact we declare all the matrices to be empty. In the second fact we allow any of the initialization operations that affect the matrices to occur in any order. This allows the matrices to be gradually filled in one operation at a time.

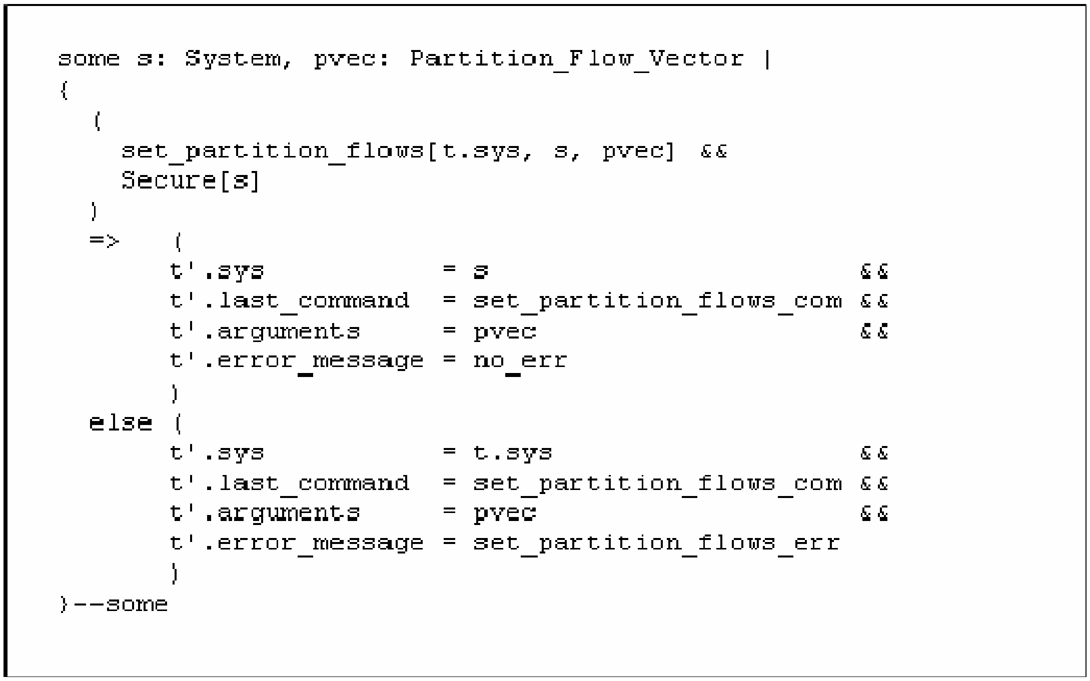
In this model we added signatures for the ResourceID and PartitionID in order to align the specification with our interface document. We added them to the runtime model also in order to maintain similarities between the models. In the initialization model we made Memory_Segments an extension of Resources. Memory_Segments have children and we placed constraints on the memory structure to be strictly hierarchical. Processes are extensions of subjects. We create a handle signature to represent a local descriptor table in a process. We added a Partition_Flow_Vector, Resource_Vector, and Partition_Resource_Vector to represent parameters of the operations.

We had one major difficulty in this model. Declaring the last command and arguments location seen in Figure 27 caused inconsistencies in the model. Moving the declarations as we did in Figure 28 removed the inconsistencies. We are unsure as to why this is.



```
some s: System, pvec: Partition_Flow_Vector |
{
  t'.last_command  = set_partition_flows_com &&
  t'.arguments     = pvec                    &&

  (
    set_partition_flows[t.sys, s, pvec] &&
    Secure[s]
  )
  =>    (
       t'.sys           = s       &&
       t'.error_message = no_err
       )
  else (
       t'.sys           = t.sys                      &&
       t'.error_message = set_partition_flows_err
       )
}--some
```

Figure 27. Previous Experimentation



```
some s: System, pvec: Partition_Flow_Vector |
{
  (
    set_partition_flows[t.sys, s, pvec]  &&
    Secure[s]
  )
  =>    (
       t'.sys           = s                       &&
       t'.last_command  = set_partition_flows_com &&
       t'.arguments     = pvec                    &&
       t'.error_message = no_err
       )
  else (
       t'.sys           = t.sys                   &&
       t'.last_command  = set_partition_flows_com &&
       t'.arguments     = pvec                    &&
       t'.error_message = set_partition_flows_err
       )
}--some
```

Figure 28. Final Experimentation

C. CONCLUSION

We set out to create a mapping between a FTLS and FSPM representation of the separation kernel. Alloy does not have built-in constructs for the mapping of a specification to a refinement of that specification. However, by incorporating the specification in the FSPM, its properties are retained in the FTLS models. To prove that operations preserve the properties of the models we used the traditional basic security theorem, which is essentially a proof by induction. In the future work section we discuss how the native Alloy tools can be used to implement specification refinement mapping.

The usability of the tool allowed us to implement much more of the model than previous efforts with theorem provers. We encountered some difficulties with the tool, but these were more attributable to the complexities of predicate logic than the Alloy tool itself. The more operations we added to the model the more computationally intensive it became for the Analyzer to process the model. Consequently the largest scope we could achieve in the initialization model was a scope of three. In this respect the current version of the Alloy toolset was inadequate. We have seven possible inputs, but the analyzer is only able to test any combination of three of them. A much larger scope would be necessary to reasonably assume that for this larger, but nonetheless limited scope, we have demonstrated the security of the model.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION AND FUTURE WORK

A. EVALUATION CRITERIA

Table 1 presents evaluation criteria for the analysis of a verification tool originally produced by Ubhayakar [Ubh03] and extended by DeCloss [Dec06] for the TCX project.

Being a model analyzer Alloy can not prove theorems. Therefore we must look at Alloy differently in order to evaluate its theorem analysis and conjecture generation abilities. Assertions in Alloy are analogous to theorems and can be verified within a limited scope as we discussed regarding the small scope hypothesis. When any counter example is found to an assertion a visual model is produced. If no counter example is found then no model is produced. This automated validation of assertions would be analogous to automated theorem proving. However, Alloy has the added advantage of allowing the user to visually inspect the model to more fully understand the inconsistency. In general the user can always visually inspect a specification by running any predicate. Therefore Alloy ranks high in the area of executable specifications.

The Alloy specification language has an incredibly low learning curve. Basic understanding of predicate logic and set theory is all that is required. People who work specifically in formal verification and computer science generally already possess a solid understanding these concepts. At most a refresher in the join operation and a familiarization of the multiplicity syntax of Alloy is required. A small set of libraries that perform common functions is provided with the default download. As the tool matures we expect a richer set of libraries to be developed. Therefore Alloy ranks extremely well regarding its specification language, usability, and extensibility.

Evaluation Criteria	***Definition***	***Utility***
Product Maturity	*A tool should be old enough and currently maintained and supported*	*Specific questions need to be answered in a timely manner regarding syntax and specification language*
Usability of Tool and Verification Environment	*The level of simplicity and flexibility of operations provided to the user*	*The interface and commands should be simple to understand and should provide syntax highlighting and error checking to increase efficiency*
Theorem Proving	*Interactive versus automated theorem proving*	*Theorem proving should be easily integrated and provide meaningful descriptions of errors and logging capabilities*
Specification Language	*Syntactical elements of the language*	*Learning curve associated with language should be minimal to provide efficient generation of specifications*
Executable Specifications	*Ability to test system directly from specification language*	*Executable specifications provide the user with a general "feel" for the system*
Multiple Levels of Abstraction	*Refinement capabilities from more abstract specifications to more concrete specifications*	*Multiple levels of abstractions provides ability to verify that the top level specification satisfies security policy*
Automatic generation of Conjectures	*Ability to automatically state items which must be proven*	*This aids in ensuring that all obligations regarding the system are being addressed*
Semantics	*Powerful expression of logic with minimal complexity*	*Underlying logic and foundational theory affects the expressiveness of the tool regarding system properties*

Table 1. Evaluation Criteria

Much like an object oriented programming language, Alloy is naturally suited for multiple levels of abstraction. With the '*extends*' and '*abstract*' keywords a specification can be reused to create more concrete models. This specification reuse almost fulfills the definition of a morphism [Spe04], but falls short. Since the demonstration of a morphism is not a built in feature there can not be automatic generation of conjectures. In the future work section we discuss how to use Alloy to completely demonstrate morphisms. Nonetheless, Alloy is naturally suited for multiple levels of abstraction and ranks well in this area.

Alloy is a semi-higher order logic system, but is not fully capable of higher ordered logic. For the most part this is not a problem. In many occasions a problem stated in higher order logic can be restate in first ordered logic. The built in operator for the transitive closure helps in avoiding higher ordered logic. Nonetheless, compared to some other tools the semantics of Alloy are technically less expressive.

Alloy is a relatively new tool, but is developing support rapidly. It was developed at Massachusetts Institute of Technology (MIT), a world renowned university. It has gained in popularity rapidly with a large support community emerging. Questions to the Alloy community discussion group are thoughtfully responded to within 24 hours. The discussion group and website are well maintained by the research students at MIT. Alloy does not require training courses, though the website contains excellent tutorials and user manuals. Alloy's conceiver and promoter Daniel Jackson wrote a textbook about Alloy titled "Software Abstractions." However, it was published only recently in 2006. The Alloy community held its first conference in November of 2006. The conference was well attended by academic, industry, and government researchers.

We would like to rank Alloy high in product maturity, but its short history prevents us from doing so. For example, during our experimentation we witnessed an update from Alloy 3 to Alloy 4. This update simplified the Alloy installation. However, the update contained some irksome syntax changes.

The Alloy Analyzer operates with in a GUI interface with one window for editing the specification and another for compilation. A button will open another window for visualizing the model in a variety of forms. Unfortunately the Alloy Analyzer GUI editor does not offer syntax highlighting. This is the one major fault of the tool. The discussion group contained dialog of users implementing syntax highlighting with the java IDE environment Eclipse. We expect that as the tool matures syntax highlighting will become a native component of the Alloy Analyzer.

Overall the tool ranks well with our evaluation criteria. Its major drawback is product maturity, but this is sure to be solved with time.

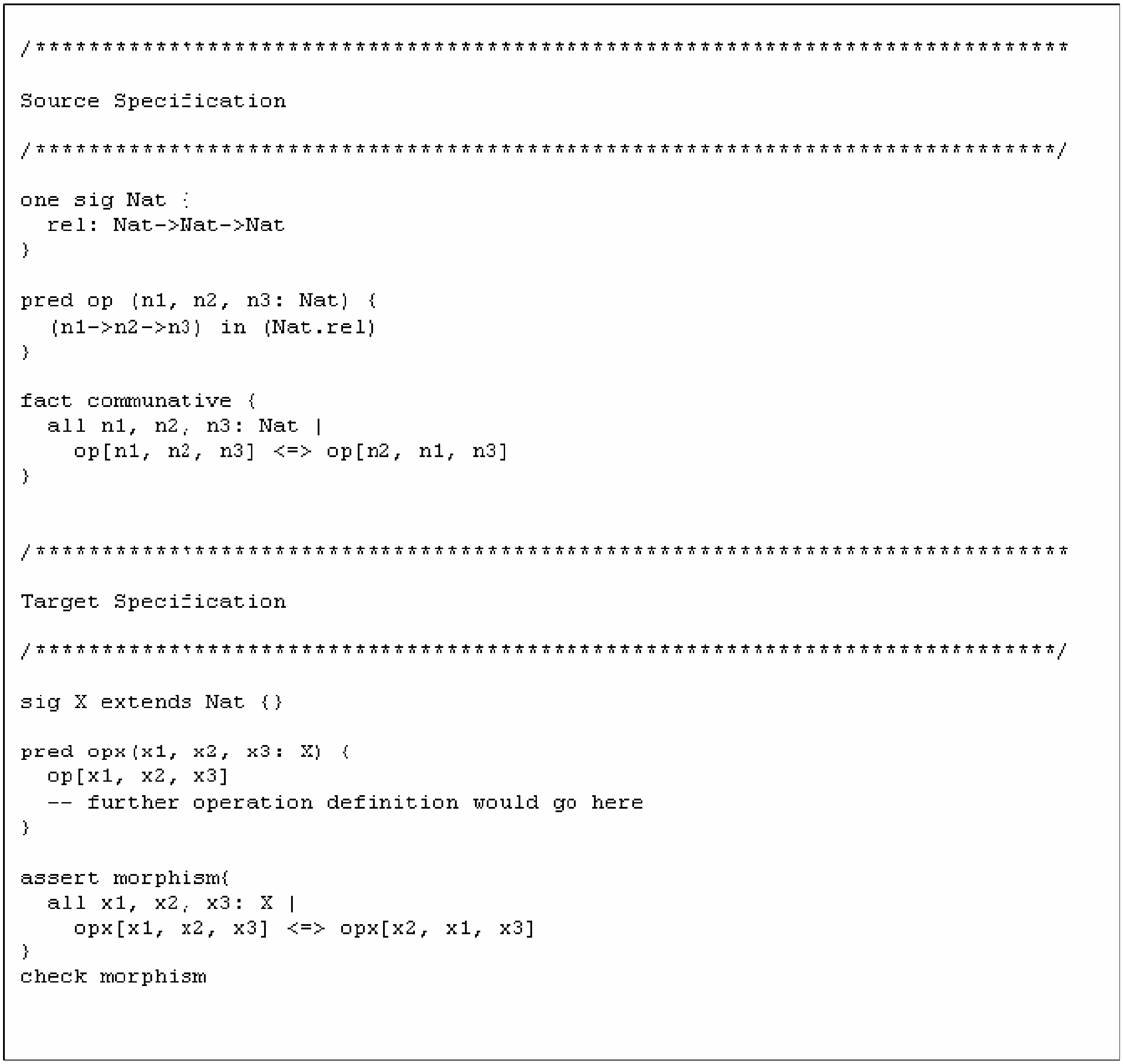
B. RESULTS

We have demonstrated that Alloy is an important tool to include in the toolbox of rapid high assurance development. Its low learning curve allows for the beginner to formal methods and high assurance systems to quickly hone their development skills. While we did not specify the FSPM and the FTLS in their entirety, this was not a fault of Alloy itself, but the time constraints of a master's level thesis. With the work we have completed so far it is reasonable to believe that an additional quarter's thesis slot would produce a richer, fuller model that includes internal resources, exported resources, time slices, and dynamic features of the TCX LPSK. Our most exciting result is that Alloy proved itself useful even for more seasoned developers.

The weakness we discovered in the TPO has proven Alloy's definite development value. Additionally, Alloy is naturally suited for demonstrating morphisms. However, it does not have built-in constructs, libraries, or templates for doing so. Our approach of including the FSPM within the FTLS to demonstrate security in the FTLS was useful to the current developers of the TCX LPSK. However, this approach does not fully satisfy the definition of morphism in the truest sense. In the future work section below we give a more precise definition of morphism and give an example of how to demonstrate one in Alloy.

C. FUTURE WORK

The Specware 4.1 Tutorial [Spe04] contains an excellent explanation of morphisms. "A morphism is a mapping from a source spec to a target spec. More precisely, it consists of two functions: one maps each type symbol of the source to a type symbol of the target, and the other maps each op symbol of the source to an op symbol of the target." Alloy is naturally suited to demonstrate morphisms by reusing specifications. The nature of extending a signature creates mapping each type symbol. Therefore, in Alloy simply by reusing a specification the first function of a morphism has been satisfied. However, there remains the effort of proving that the properties of the source specification operations are retained in the target specification operations. Figure 29 demonstrates how to do this in Alloy using the natural numbers and commutativity example given in the Specware tutorial. For future work we recommend further investigating this approach. The level of difficulty and effectiveness in using this approach with the TCX LPSK would be an interesting analysis for the TCX project.



```
/******************************************************************************

Source Specification

/*****************************************************************************/

one sig Nat {
  rel: Nat->Nat->Nat
}

pred op (n1, n2, n3: Nat) {
  (n1->n2->n3) in (Nat.rel)
}

fact communative {
  all n1, n2, n3: Nat |
    op[n1, n2, n3] <=> op[n2, n1, n3]
}


/******************************************************************************

Target Specification

/*****************************************************************************/

sig X extends Nat {}

pred opx(x1, x2, x3: X) {
  op[x1, x2, x3]
  -- further operation definition would go here
}

assert morphism{
  all x1, x2, x3: X |
    opx[x1, x2, x3] <=> opx[x2, x1, x3]
}
check morphism
```

Figure 29. Morphism With Alloy

APPENDIX A: SEPARATION KERNEL FSPM IN ALLOY

```
/*****************************************************************************

 A Least Privilege Model for Static Separation Kernels
  from  T.Levin, C.Irvine, T.Nguyen, CISR Tech Report NPS-CS-05-003
 October 2004
 (references to the Tables in the paper)

*****************************************************************************/

module sep_kernel

-- library containing the dom function
open util/ternary as tern


/*****************************************************************************

The information in computer systems is often thought of generically as
resources.  The flow of information is then described as a matrix of what
resource is allowed to read and/or write to another resource.  This is
referred to as an access matrix.  A resource is anything in the system we wish
to consider, be it a file, a process, a user, a program, firmware, etc.  In
Alloy a 3-tuple or 3 item relation can be used to indicated the access matrix
like so, Resource -> Resource -> Mode  The 'abstract' keyword below indicates
that an item called Mode should not be created in the model.  The 'extends' and
'abstract' keywords are analogous to object oriented programming with a
heirarchy of parent and child classes.  The 'one' keyword means one and only
one item called RD and one and only one item called WT should be the model.

*****************************************************************************/

-- R in the paper
sig Resource
{
-- Every Resource belongs to exactly one Block, no more, no less
  master: one Block
}

-- access rights type, F in the paper
abstract sig Mode {}

-- read and write in the Paper
```

```
one sig RD, WT extends Mode {}


/******************************************************************************

A separation kernel partitions all the resources.  The separation kernel is
simplistic in that it only provides the isolation of resources into partitions
which we will call Blocks.  In the system every Resource will be assigned
exactly one and only one Block, thus the Blocks represent equivalence classes
of Resources.  Our separation kernel will be static, therefore there will be
no reassigning of Resources to different Blocks once the kernel is running and
it will not make sense to have empty Blocks.  It is not useful to have all
resources partitioned into a single Block so that case may be explicitly
excluded from the model in order to make it more interesting.

******************************************************************************/

-- B in the paper
sig Block {}
-- each block has at least one resource, no empty blocks
{
  some r: Resource | r.master = this
}

-- a minimally useful kernel, this fact may be commented out
fact { #Block > 1 }


/******************************************************************************

The system will contain two matrices.  One to represent the flow of information
between blocks and one for the flow of information between resources.  The two
matrices are intended to orthogonal, that is, there may be flows allowed in one
that are not allowed in the other.  We will be interested in taking the partial
ordering and the trusted partial ordering of the block flow matrix.  To help
with the trusted partial ordering we create the structure below.

We can consider a read originating from BlockA to BlockB to be the same as a
write originating from BlockB to BlockA in terms of information flow.  We can
simplify either statement by saying information has flowed from BlockB to
BlockA.

Alloy has a built in command ^ (carrot) for constructing transitive closures.
The transitive closure is useful for seeing all blocks that information can
eventually flow into.
```

```
******************************************************************************/

-- Block-To-Block Flow Matrix, Table 1
sig BB{
  flow:        Block -> Block -> Mode,

-- these are secondary, derived from the flow
  basic_flow:  Block -> Block,
  FLOWS:       Block -> Block
}
{
-- definition of basic flow
  basic_flow =
  {
    a, b: Block |
      WT in flow[a][b] or
      RD in flow[b][a]
  }

-- FLOWS is a transitive closure of basic_flow
  FLOWS = ^ basic_flow

-- These two are for better visualizing and more interesting models

-- block always can access itself with Read/Write
  all b: Block | #b.(b.flow) = #Mode

-- each access mode is represented at least once in the elements of flow
    RD in Block.(Block.flow)
    WT in Block.(Block.flow)
}


/******************************************************************************

The word subject and object are typically used to indicate which resource is
the active entity performing the read and/or write on another passive resource.
In our model we use the Subject terminology and we allow one Subject to
operate on another Subject (ie. one process communicating to another process).
We drop the Object nomenclature and simply refer to a Subject accessing a
Resource (which may be another Subject).

******************************************************************************/
```

```
sig Subject extends Resource {}


/******************************************************************************

Our system is really rather simple.  It is two access matrices.  One is the
block flow (bb) and the other the least privilege matrix (sr_flow) which
constrains the block flow (bb).  Later, we will define a secure system to be a
system that only allows flows that are permitted by both matrices.  Technically
sr_flow is not required to be a subset of bb.  That is, it may allow flows that
are prohibited by bb, however, in a secure system those flows would never be
realized, because what is allowed is only the intersection of the two matrices.

A Block may contain any combination of Subjects and Resources.  Therefore, the
Block nomenclature is devoid of the Subject and Resource distinction.  In
order to better visualize and talk about the flow of information only Subjects
are allowed to read and/or write.  That is Resources that are not Subjects
should never initiate a flow.

Of course we are only interested in Resources and Blocks that are inside of our
System.  Sense all Resources belong to a Block, Alloy will not produce
Resources or Blocks that have nothing to do with each other.  In like manner we
need to instruct Alloy that every block participates in the System to prevent
Alloy from producing models with Resources and Blocks that are outside of the
System.

MM should be orthogonal to SR and BB, that is SR/BB is the policy of what is
allowed, and MM is what the program wants to do.  The security predicate will
say that a secure system allows only the program actions that conform to the
policy.

It is important to note that information is not required to flow into or out of
a Block.  However, a completely isolated Block would not be useful.

******************************************************************************/

sig System {
  bb:        BB,
  sr_flow:   Subject->Resource->Mode,
  MM:        Subject -> Resource -> Mode
}
{
-- Every Block participates in the System.
  Block in Block.((bb.flow).Mode) + ((bb.flow).Mode).Block
-- sr_flow is non empty
  some sr_flow

/*
&&
-- This constraint is only to create interesting models.  It is recommend to
-- comment out this section to study the model the way it was intended.
-- A completely isolated Block is not useful therefore:
  -- Every block,
  all b1: blocks |
    -- has some other block,
    some b2: Block |
      -- that is not itself,
      disj[b1,b2]  &&
      (
        -- which it neither flows into,
        b2 in b1.flow or
        -- nor out of.
        b2 in flow.b1
      )
*/
}


/******************************************************************************

We are interested with a traditional confidentiality policy between the Blocks
that prohibits reading up or writing down.  The separation kernel itself only
isolates the Blocks and regulates the flow between them, but is devoid of
semantics labeling one Block greater than another.

However, we can define a predicate that will require the Blocks to be
partitioned in such a way that information is not allowed to flow circularly.
That is if information leaves BlockA, there is no transitive closure of the
flows that will lead back to BlockA. It is important to note that any two
Blocks are not required to be related.  For example, BlockA->BlockC,
BlockB->BlockC, but there is no flow between BlockA and BlockB and no
indication if one is greater than the other.  Therefore such a non-circular
flow of information is technically considered a partial ordering but not a
total ordering.  That is, not all the items in the set are comparable.

Mathematically a partial ordering (PO) is defined as a relation over a set that
is transitive and antisymmetric.  Antisymmetric means there is no circularity
in the relation.  Many authors extend a partial ordering to include reflexivity
which means every item in the set is comparable to itself.  In our situation
```

```
this may or may not be the case.  We include reflexivity in PO allowing it to
be commented out if not desired.

Note: A lattice is a partial ordering where every comparable pair of items
contain both a least upper bound (lub) and a greatest lower bound (glb), which
results in a universal upper and lower bound to the lattice.

******************************************************************************/

pred PO(bb: Block->Block){
  all  i,j,k: (bb.Block + Block.bb) |

-- reflexive
    (
      i->i in bb
    ) &&

-- antisymmetric
    (
      ((i->j in bb) && (j->i in bb)) => (i=j)
    ) &&

-- transitive
    (
      ((i->j in bb)  && (j->k in bb)) => (i->k in bb)
    )
}


/******************************************************************************

Overtime an item of information may no longer be sensitive and its sensitivity
may need to be downgraded.  To do so requires violating the partial ordering.
Therefore the notion of a Trusted Subject is introduced.  A Trusted Subject is
a Subject (ie. process) that has undergone rigorous analysis and is trusted not
to downgrade information other than the information it is intended to
downgrade.

******************************************************************************/

sig Trusted_Subject extends Subject {}


/******************************************************************************
```

A Trusted Subject is allowed to violate the partial ordering but it is not required to violate the partial ordering. Therefore, there may be flows in the System that are the result of the Trusted Subjects and violate the partial ordering. The problem here is that we do not really have a way of identifying which Subject, trusted or otherwise, caused a flow in bb, after all the matrices are intended to be orthogonal. However, the reverse may be possible. We can identify the flows in sr_flow and require any flow that upsets the partial ordering in bb to be a trusted subject. However, this also requires us to identify the partial ordering.

```
******************************************************************************/

pred TPO(sys: System){
  let Nontrusted_Subs_in_SR = dom[sys.sr_flow] - Trusted_Subject,

      Nontrusted_Block_Flow = { b1, b2: Block , m: Mode |
                 (
        some sub: Nontrusted_Subs_in_SR, r: Resource |
        (
          -- sub is a non-trusted subject in the subject part of sr_flow
          -- when combined with some resource and
          -- and the mode of Nontrusted_Block_Flow is also in sr_flow
          (sub -> r -> m) in sys.sr_flow and

          -- and the corresponding blocks of that subject and resource
          -- comprises the blocks of Nontrusted_Block_Flow
          b1 = sub.master and b2 = r.master
        )
      )
      } |

  -- The transitive closure of
  -- the intersection of the Nontrusted_Block_Flow and bb.flow
  -- with the Mode removed
  -- should be a partial ordering
  PO[ ^((Nontrusted_Block_Flow & sys.bb.flow).Mode) ]
}


/******************************************************************************
```

A secure system is a system that is a trusted partial ordering where only flows in by both matrices (sr_flow and bb.flow) are allowed.

```
******************************************************************************/
```

```
pred Secure(sys: System)
{
  TPO[sys]  &&
  all sub: Subject, res: Resource, mod: Mode |
  (
    (sub -> res -> mod) in sys.MM

    =>

    (
      (sub -> res -> mod) in sys.sr_flow &&
      mod in sys.bb.flow[sub.master][res.master]
    )
  )
}


pred show () {}
run show
run show for 3 but exactly 3 Block
run show for 4 but exactly 4 Block
run show for 3 but exactly 3 Block, exactly 6 Resource

pred showPO () { some sys: System | PO[sys.bb.FLOWS] }
run showPO
run showPO for 3 but exactly 3 Block
run showPO for 4 but exactly 4 Block
run showPO for 3 but exactly 3 Block, exactly 6 Resource

pred showTPO () { some sys: System | TPO[sys] }
run showTPO
run showTPO for 4 but exactly 4 Block, exactly 1 Trusted_Subject

pred showSecure() {
  some sys: System | Secure[sys]
}
run Secure
run showSecure
run showSecure for 4 but exactly 4 Block, exactly 1 Trusted_Subject
run showSecure for 4 but exactly 4 Block, exactly 0 Trusted_Subject
```

APPENDIX B: SEPARATION KERNEL INIT FTLS IN ALLOY

```
/******************************************************************************

 A Least Privilege Model for Static Separation Kernels
  from  T.Levin, C.Irvine, T.Nguyen, CISR Tech Report NPS-CS-05-003
 October 2004
 (references to the Tables in the paper)

 Least Privilege Separation Kernel Interface
  working note Version 0.6
 12 February 2007

******************************************************************************/

module sep_kernel_init

-- library containing the dom function
open util/ternary as tern

-- For state transitions
open util/sequence[Transition] as set_up_seq


/******************************************************************************

The premilinary interface uses parameters such as resource_id and partition_id
to the interface functions.  In order to conform as much as possible to the
interace we have added these to the to Resource and Block.  The Alloy Analyzer
will place an integer identifier for each Resource, Block, etc. that it
created in a model.  Therefore, adding the Resource_ID and Partition_ID
constructs may not be entirely necessary, but creates for a stronger mapping.

disj is a native Alloy command meaning disjoint.  It used here to require two
elements to be distinct.

******************************************************************************/

-- Unique Resource identifier
sig Resource_ID {}

-- Every resource has a unique ID
fact
{
  all r1, r2: Resource |
    disj[r1,r2] => disj[r1.resource_id, r2.resource_id]
}

-- garuntees no resource_id's that do no participate in the universe
fact
{
  Resource_ID in Resource.resource_id
}

-- Retrieve the corresponding Resource
-- This function is not currently used
fun id_to_resource (i: Resource_ID): set Resource
{
  {

    r: Resource | r.resource_id = i
  }
}


-- Unique Block identifier
sig Partition_ID {}

-- Every block has a unique ID
fact
{
  all b1, b2: Block |
    disj[b1,b2] => disj[b1.partition_id, b2.partition_id]
}

-- garuntees no parition_id's that do no participate
fact
{
  Partition_ID in Block.partition_id
}

-- Retrieve the corresponding Block
-- This function is not currently used
fun id_to_block (i: Partition_ID): set Block
{
  {
  b: Block | b.partition_id = i
  }
}
```

```
/******************************************************************************

The information in computer systems is often thought of generically as
resources.  The flow of information is then described as a matrix of what
resource is allowed to read and/or write to another resource.  This is
referred to as an access matrix.  A resource is anything in the system we wish
to consider, be it a file, a process, a user, a program, fireware, etc.  In
Alloy a 3-tuple or 3 item relation can be used to indicated the access matrix
like so, Resource -> Resource -> Mode  The 'abstract' keyword below indicates
that an item called Mode should not be created in the model.  The 'extends' and
'abstract' keywords are analogous to object oriented programming with a
heirarchy of parent and child classes.  The 'one' keyword means one and only
one item called RD and one and only one item called WT should be the model.

******************************************************************************/

-- R in the paper
sig Resource
{
-- Every Resource belongs to exactly one Block, no more, no less
  master: one Block,

-- Added to conform to the interface
  resource_id: one Resource_ID
}

-- access rights type, F in the paper
abstract sig Mode {}

-- read and write in the Paper
one sig RD, WT extends Mode {}


/******************************************************************************

A separation kernel partitions all the resources.  The separation kernel is
simplistic in that it only provides the isolation of resources into partitions
which we will call Blocks.  In the system every Resource will be assigned
exactly one and only one Block, thus the Blocks represent equivalence classes
of Resources.  Our separation kernel will be static, therefore there will be
no reassigning of Resources to different Blocks once the kernel is running and
it will not make sense to have empty Blocks.  It is not useful to have all
resources partitioned into a single Block so that case may be explicitly
```

```
excluded from the model in order to make it more interesting.

******************************************************************************/

-- B in the paper
sig Block
{
-- Added to conform to the interface
  partition_id: Partition_ID
}
-- each block has at least one resource, no empty blocks
{
  some r: Resource | r.master = this
}

-- a minimally useful kernel, this fact may be commented out
--fact { #Block > 1 }


/******************************************************************************

The system will contain two matrices.  One to represent the flow of information
between blocks and one for the flow of information between resources.  The two
matrices are intended to orthogonal, that is, there may be flows allowed in one
that are not allowed in the other.  We will be interested in taking the partial
ordering and the trusted partial ordering of the block flow matrix.  To help
with the trusted partial ordering we create the structure below.

We can consider a read originating from BlockA to BlockB to be the same as a
write originating from BlockB to BlockA in terms of information flow.  We can
simplify either statement by saying information has flowed from BlockB to
BlockA.

Alloy has a built in command ^ (carrot) for constructing transitive closures.
The transitive closure is useful for seeing all blocks that information can
eventually flow into.

******************************************************************************/

-- Block-To-Block Flow Matrix, Table 1
sig BB{
  flow:        Block -> Block -> Mode,

-- those are secondary, derived from the flow
  basic_flow:  Block -> Block,
  FLOWS:       Block -> Block
}
{
-- definition of basic flow
  basic_flow =
  {
    a, b: Block |
      WT in flow[a][b] or
      RD in flow[b][a]
  }

-- FLOWS is a transitive closure of basic_flow
  FLOWS = ^ basic_flow

-- These two are for better visualizing and more interesting models

-- block always can access itself with Read/Write
--  all b: Block | #b.(b.flow) = #Mode

-- each access mode is represented at least once in the elements of flow
--    RD in Block.(Block.flow)
--    WT in Block.(Block.flow)
}


/*******************************************************************************

The word subject and object are typically used to indicate which resource is
the active entity performing the read and/or write on another passive resource.
In our model we use the Subject terminology and we allow one Subject to
operate on another Subject (ie. one process communicating to another process).
We drop the Object nomenclature and simply refer to a Subject accessing a
Resource (which may be another Subject).

*******************************************************************************/

sig Subject extends Resource {}


/*******************************************************************************

Memory Segments are resources, but Subjects are not Memory Segments.  Therefore
subjects and Memory Segments partition Resources.  Memory Segments will be
assigned to each ring of a Process.  create_memory_obj and open_memory_obj will
need to ensure memory segments exist in a strict hierarchy.  That is a memory
```

segment will have no more than one parent and there is not looping in the hierarchy.

```
******************************************************************************/

-- A Memory_Segment is a Resource and it has zero or more childern that are
-- also Memory_Segments
sig Memory_Segment extends Resource
{
  childern: set Memory_Segment
}
{
  strongly_hierarchical[this]
}

-- No looping, and no more than one parent
pred strongly_hierarchical (m: Memory_Segment)
{

  let child = { x, y: Memory_Segment | y in x.childern },
      descendants = ^child |

  -- no loops
  -- a memory segment is not a descendant of itself
  ( not (m in m.descendants)  &&

  -- no sharing
  -- a memory segment has only one parent
  -- no two parents have the same child
  (all disj m1, m2: Memory_Segment |
       no (m1.childern & m2.childern))
  )
}


/******************************************************************************
```

Our system is really rather simple. It is two access matrices. One is the block flow (bb) and the other the least privilege matrix (sr_flow) which constrains the block flow (bb). Later, we will define a secure system to be a system that only allows flows that are permitted by both matrices. Technically sr_flow is not required to be a subset of bb. That is, it may allow flows that are prohibited by bb, however, in a secure system those flows would never be realized, because what is allowed is only the intersection of the two matrices.

```
A Block may contain any combination of Subjects and Resources.  Therefore, the
Block nomenclature is devoid of the Subject and Resource distinction.  In
order to better visualize and talk about the flow of information only Subjects
are allowed to read and/or write.  That is Resources that are not Subjects
should never initiate a flow.

Of course we are only interested in Resources and Blocks that are inside of our
System.  Sense all Resources belong to a Block, Alloy will not produce
Resources or Blocks that have nothing to do with each other.  In like manner we
need to instruct Alloy that every block participates in the System to prevent
Alloy from producing models with Resources and Blocks that are outside of the
System.

MM should be orthogonal to SR and BB, that is SR/BB is the policy of what is
allowed, and MM is what the program wants to do.  The security predicate will
say that a secure system allows only the program actions that conform to the
policy.

It is important to note that information is not required to flow into or out of
a Block.  However, a completely isolated Block would not be useful.

******************************************************************************/

sig System {
  resources: set Resource,

  bb: BB,
  sr_flow: Subject->resources->Mode,
  MM: Subject -> Resource -> Mode

  -- This a runtime extension of the model.
  -- RM: resources -> Memory_Segment
}
{
-- Every Block participates in the System.
-- We start with an empty matrix and then we add blocks
--  Block in Block.((bb.flow).Mode) + ((bb.flow).Mode).Block

-- sr_flow is non empty
--  some sr_flow

/*
&&
-- This constraint is only to create interesting models.  It is recommend to
-- comment out this section to study the model the way it was intended.
-- A completely isolated Block is not useful therefore:
  -- Every block,
  all b1: blocks |
    -- has some other block,
    some b2: Block |
      -- that is not itself,
      disj[b1,b2]  &&
      (
        -- which it neither flows into,
        b2 in b1.flow or
        -- nor out of.
        b2 in flow.b1
      )
*/
}


/******************************************************************************

We are interested with a traditional confidentiality policy between the Blocks
that prohibits reading up or writing down.  The separation kernel itself only
isolates the Blocks and regulates the flow between them, but is devoid of
semantics labeling one Block greater than another.

However, we can define a predicate that will require the Blocks to be
partitioned in such a way that information is not allowed to flow circularly.
That is if information leaves BlockA, there is no transitive closure of the
flows that will lead back to BlockA. It is important to note that any two
Blocks are not required to be related.  For example, BlockA->BlockC,
BlockB->BlockC, but there is no flow between BlockA and BlockB and no
indication if one is greater than the other.  Therefore such a non-circular
flow of information is technically considered a partial ordering but not a
total ordering.  That is, not all the items in the set are comparable.

Mathematically a partial ordering (PO) is defined as a relation over a set that
is transitive and antisymmetric.  Antisymmetric means there is no circularity
in the relation.  Many authors extend a partial ordering to include reflexivity
which means every item in the set is comparable to itselfs.  In our situation
this may or may not be the case.  We include reflexivity in PO allowing it to
be commented out if not desired.

Note: A lattice is a partial ordering where every comparable pair of items
contain both a least upper bound (lub) and a greatest lower bound (glb), which
results in a universal upper and lower bound to the lattice.
```

```
******************************************************************************/

pred PO(bb: Block->Block){
  all  i,j,k: (bb.Block + Block.bb) |

-- reflexive
    (
      i->i in bb
    ) &&

-- antisymmetric
    (
      ((i->j in bb) && (j->i in bb)) => (i=j)
    ) &&

-- transitive
    (
      ((i->j in bb)  && (j->k in bb)) => (i->k in bb)
    )
}


/******************************************************************************

Overtime an item of information may no longer be sensitive and its sensitivity
may need to be downgraded.  To do so requires violating the partial ordering.
Therefore the notion of a Trusted Subject is introduced.  A Trusted Subject is
a Subject (ie. process) that has undergone rigorous analysis and is trusted not
to downgrade information other than the information it is intended to
downgrade.

******************************************************************************/

sig Trusted_Subject extends Subject {}


/******************************************************************************

A Trusted Subject is allowed to violate the partial ordering but it is not
required to violate the partial ordering.  Therefore, there may be flows in the
System that are the result of the Trusted Subjects and violate the partial
ordering.  The problem here is that we do not really have a way of identifying
which Subject, trusted or otherwise, caused a flow in bb, after all the
matrices are intended to be orthogonal.  However, the reverse may be possible.
We can identify the flows in sr_flow and require any flow that upsets the
```

```
partial ordering in bb to be a trusted subject.  However, this also requires us
to identify the partial ordering.

******************************************************************************/

pred TPO(sys: System){
  let Nontrusted_Subs_in_SR = dom[sys.sr_flow] - Trusted_Subject,

      Nontrusted_Block_Flow = { b1, b2: Block , m: Mode |
                 (
        some sub: Nontrusted_Subs_in_SR, r: Resource |
        (
          -- sub is a non-trusted subject in the subject part of sr_flow
          -- when combined with some resource and
          -- and the mode of Nontrusted_Block_Flow is also in sr_flow
          (sub -> r -> m) in sys.sr_flow and

          -- and the corresponding blocks of that subject and resource
          -- comprises the blocks of Nontrusted_Block_Flow
          b1 = sub.master and b2 = r.master
        )
      )
      } |

  -- The transitive closure of
  -- the intersection of the Nontrusted_Block_Flow and bb.flow
  -- with the Mode removed
  -- should be a partial ordering
  PO[ ^((Nontrusted_Block_Flow & sys.bb.flow).Mode) ]
}


/******************************************************************************

A secure system is a system that is a trusted partial ordering where only flows
in by both matrices (sr_flow and bb.flow) are allowed.

******************************************************************************/

pred Secure(sys: System)
{
  TPO[sys]  &&
  all sub: Subject, res: Resource, mod: Mode |
  (
    (sub -> res -> mod) in sys.MM
    =>

    (
      (sub -> res -> mod) in sys.sr_flow &&
      mod in sys.bb.flow[sub.master][res.master]
    )
  )
}


/******************************************************************************

The above defines a secure system and what it means to be in a secure state.
In the strictest sense the concept of transitioning from one state to another
does not exist in this definition because the system is static and either the
system is secure or it isn't.  The matrices do not change once brought into
existence.

This model represents the initialization phase of the system.  We start with
an empty system (no matrices with no allowed flows) and end with the matrices
that conform to the security predicate.

In order to visualize transitions operation we create a Transition signature
that includes the current system along the last command execute, the
corresponding arguments, and the last error message.

We create stubs for many of the init commands.  They may not effect our
definition of security so for now we leave them out of the model.

******************************************************************************/

-- List of commands in the preliminary interface.
abstract sig Command {}
one sig no_op_com,
    set_partition_flows_com,
    set_resource_flows_com,
    create_partition_com,
    create_process_com,
    create_memory_object_com,
    open_memory_object_com,
    close_memory_object_com
extends Command {}

-- An error for each command
```

```
abstract sig Error { }
one sig no_err,
    set_partition_flows_err,
    set_resource_flows_err,
    create_partition_err,
    create_process_err,
    create_memory_object_err,
    open_memory_object_err,
    close_memory_object_err
extends Error {}


sig Transition
{
  error_message: Error,
  last_command:  Command,
  arguments:     set univ,

  sys: System
}


/******************************************************************************

Parameters for some of the commands.

******************************************************************************/

-- Used by set_partition_flows
sig Partition_Flow_Vector
{
  v: Partition_ID -> Partition_ID -> Mode
}
{
  some v
}

-- Used by set_resource_flows
sig Resource_Vector
{
  v: Resource_ID -> Resource_ID -> Mode
}
{
  some v
}
```

```
-- Used by create_partition
sig Partition_Resource_Vector
{
  v: Partition_ID -> Resource_ID
}
{
  some v
}


/*******************************************************************************

Create Process, create_mem_obj, open_mem_obj all take handles as input.

*******************************************************************************/

sig Handle
{
  seg: one Memory_Segment,
  mod: Mode
}


/*******************************************************************************

We define a process here because this is the first time it is introduced to the
model.  It has a time slice and set of memory segments, but for now we leave
those out of the model.

*******************************************************************************/

sig Process extends Subject
{
  ring1: one Memory_Segment,
  ring2: one Memory_Segment,
  ring3: one Memory_Segment,

  -- Local Descriptor Table
  ldt: set Handle
}
{
  -- The ring handles point to distinct memory
  disj[ring1, ring2, ring3]
}
```

```
/******************************************************************************

The first tranistion in the program begins empty, with no commands, arguments,
or error messages.  The matrices are entirely empty.

******************************************************************************/

fact init_set_up_seq{
  let t = Seq::set_up_seq/first[] |

    no_err    = t.error_message &&
    no_op_com = t.last_command &&
    no t.arguments &&

    no t.sys.resources &&
    no t.sys.bb.flow   &&
    no t.sys.sr_flow   &&
    no t.sys.MM
}


/******************************************************************************

Every transition involves a command in the interface document.  As the
commands are executed either error messages are generated or the matrices are
filled in.  The || (or) condition separate each possible command.

******************************************************************************/

fact trans_setup
{
  -- This is for more interesting models
  --set_up_seq/allExistNoDuplicates[] &&

  -- For all seqence indices except the first one
  all sqidx: Seq::set_up_seq/inds[] - set_up_seq/firstIdx[] |
  {

   let t = set_up_seq/Seq::at[ set_up_seq/prev[sqidx] ],
       t'= set_up_seq/Seq::at[sqidx]
    |

    -- To prevent a tranisition that is the same command, same args,
    -- same error and same system from occuring next to each other
    --disj[t'.sys, t.sys] &&


    -- sequence of commands (predicates) separated by  or

    some s: System, pvec: Partition_Flow_Vector |
    {
      (
        set_partition_flows[t.sys, s, pvec] &&
        Secure[s]
      )
        =>   (
              t'.sys           = s                       &&
              t'.last_command  = set_partition_flows_com &&
              t'.arguments     = pvec                    &&
              t'.error_message = no_err
             )
        else (
              t'.sys           = t.sys                   &&
              t'.last_command  = set_partition_flows_com &&
              t'.arguments     = pvec                    &&
              t'.error_message = set_partition_flows_err
             )
    }--some

    ||

    some s: System, rvec: Resource_Vector |
    {
      (
        set_resource_flows[t.sys, s, rvec] &&
        Secure[s]
      )
        =>   (
              t'.sys           = s                      &&
              t'.last_command  = set_resource_flows_com &&
              t'.arguments     = rvec                   &&
              t'.error_message = no_err
             )
        else (
              t'.sys           = t.sys                  &&
              t'.last_command  = set_resource_flows_com &&
              t'.arguments     = rvec                   &&
              t'.error_message = set_resource_flows_err
             )
    }--some

    ||

    some s: System, pvec: Partition_Resource_Vector |
    {
      (
        create_partition[t.sys, s, pvec] &&
        Secure[s]
      )
        =>   (
              t'.sys           = s                      &&
              t'.last_command  = create_partition_com   &&
              t'.arguments     = pvec                   &&
              t'.error_message = no_err
             )
        else (
              t'.sys           = t.sys                  &&
              t'.last_command  = create_partition_com   &&
              t'.arguments     = pvec                   &&
              t'.error_message = create_partition_err
             )
    }--some

    ||

    some s: System, process_id: Resource_ID, part_id: Partition_ID,
                    hd1, hd2, hd3: Handle  |
    {
      disj[hd1, hd2, hd3] &&

      (
        create_process[t.sys, s, process_id, part_id, hd1, hd2, hd3] &&
        Secure[s]
      )
        =>   (
              t'.sys           = s                      &&
              t'.last_command  = create_process_com     &&
              t'.arguments     = process_id + part_id +
                                 hd1 + hd2 + hd3        &&
              t'.error_message = no_err
             )
        else (
              t'.sys           = t.sys                  &&
              t'.last_command  = create_process_com     &&
              t'.arguments     = process_id + part_id +
                                 hd1 + hd2 + hd3        &&
              t'.error_message = create_process_err
             )
    }--some

    ||

    some s: System, parent_handle: Handle, part_id: Partition_ID |
    {
      (
        create_memory_object[t.sys, s, parent_handle, part_id] &&
        Secure[s]
      )
        =>   (
              t'.sys           = s                         &&
              t'.last_command  = create_memory_object_com  &&
              t'.arguments     = parent_handle + part_id   &&
              t'.error_message = no_err
             )
        else (
              t'.sys           = t.sys                     &&
              t'.last_command  = create_memory_object_com  &&
              t'.arguments     = parent_handle + part_id   &&
              t'.error_message = create_memory_object_err
             )
    }--some

    ||

    some s: System, parent_handle, obj_handle: Handle, mod: Mode |
    {
      (
        open_memory_object[t.sys, s, parent_handle, obj_handle, mod] &&
        Secure[s]
      )
        =>   (
              t'.sys           = s                                &&
              t'.last_command  = open_memory_object_com           &&
              t'.arguments     = parent_handle + obj_handle + mod &&
              t'.error_message = no_err
             )
        else (
              t'.sys           = t.sys                            &&
              t'.last_command  = open_memory_object_com           &&
              t'.arguments     = parent_handle + obj_handle + mod &&
              t'.error_message = open_memory_object_err
             )
    }--some

    ||

    some s: System, obj_handle: Handle |
    {
      (
        close_memory_object[t.sys, s, obj_handle] &&
        Secure[s]
      )
        =>   (
              t'.sys           = s                       &&
              t'.last_command  = close_memory_object_com &&
              t'.arguments     = obj_handle              &&
              t'.error_message = no_err
             )
        else (
              t'.sys           = t.sys                   &&
              t'.last_command  = close_memory_object_com &&
              t'.arguments     = obj_handle              &&
              t'.error_message = close_memory_object_err
             )
    }--some

  }--all

}


/******************************************************************************

There is one predicate here for each command.

******************************************************************************/

pred set_partition_flows(sys, sys': System, pvec: Partition_Flow_Vector)
{
  let new_bb =
  {
    b1, b2: Block, m: Mode |
    (
      (b1.partition_id -> b2.partition_id -> m) in pvec.v
    )
  } |

  sys'.bb.flow = sys.bb.flow + new_bb &&

  sys'.sr_flow = sys.sr_flow          &&
  sys'.MM = sys.MM                    &&
  sys'.resources = sys.resources
}


pred set_resource_flows(sys, sys': System, rvec: Resource_Vector)
{
  let new_sr_flow =
  {
    s: Subject, r: Resource, m: Mode |
    (
      (s.resource_id -> r.resource_id -> m) in rvec.v
    )
  } |

  sys'.sr_flow = sys.sr_flow + new_sr_flow       &&
  sys'.bb = sys.bb                               &&
  sys'.MM = sys.MM                               &&
  sys'.resources = sys.resources +
                   (new_sr_flow.Mode).Resource /* +
                   Subject.(new_sr_flow.Mode)    */
}


-- For a future dynamic extension of the model
-- pred set_subj_privileges(sys, sys': System, pvec: Privilege_Vector) {}


pred create_partition(sys, sys': System, pvec: Partition_Resource_Vector)
{

  -- Every resource that corresponds to the resource_id in the vector
  -- belongs to the systems set of resources
  sys'.resources = sys.resources +
                   {r: Resource | r.resource_id in Partition_ID.(pvec.v)} &&


  -- all the old blocks are in the new system
  -- all the added blocks are in the new system
  let add_blocks = {b: Block | b.partition_id in (pvec.v).Resource_ID},
      old_blocks = (Block.((sys.bb.flow).Mode) + ((sys.bb.flow).Mode).Block),
      new_blocks = (Block.((sys'.bb.flow).Mode) + ((sys'.bb.flow).Mode).Block)
  |

  (old_blocks + add_blocks) in new_blocks


  -- everything else stays the same
  sys'.sr_flow = sys.sr_flow &&
  sys'.MM = sys.MM
}


pred create_process(sys, sys': System,
                    process_id: Resource_ID, part_id: Partition_ID,
                    handle1, handle2, handle3: Handle)
{
-- Assign process_id to MM, etc

  -- add process
  some proc: Process |
    not ( proc in sys.resources  ) &&
        ( proc in sys'.resources ) &&

    proc.resource_id         = process_id &&
    proc.master.partition_id = part_id    &&


    -- makes sure it has the right properties
    proc.ring1 = handle1.seg  &&
    proc.ring2 = handle2.seg  &&
    proc.ring3 = handle3.seg
}


pred create_memory_object(sys, sys': System,
                          parent_obj_handle: Handle,
                          part_id: Partition_ID)
{
  some child: Memory_Segment |
  {
    child in parent_obj_handle.seg.childern      &&
    part_id in child.master.partition_id         &&
--    strongly_hierarchical[parent_obj_handle.seg] &&

    sys'.resources = sys.resources + child
  }

  sys'.bb.flow = sys.bb.flow     &&
  sys'.sr_flow = sys.sr_flow     &&
  sys'.MM      = sys.MM
}

pred open_memory_object(sys, sys': System,
                        parent_obj_handle, obj_handle: Handle,
                        md: Mode)
{
  some child: parent_obj_handle.seg.childern |
  {
    child in obj_handle.seg  &&
    md    in obj_handle.mod
  }


  sys'.bb.flow = sys.bb.flow     &&
  sys'.sr_flow = sys.sr_flow     &&
  sys'.MM      = sys.MM          &&
  sys'.resources = sys.resources
}

pred close_memory_object(sys, sys': System,
                         obj_handle: Handle)
{
  sys'.bb.flow = sys.bb.flow     &&
  sys'.sr_flow = sys.sr_flow     &&
  sys'.MM      = sys.MM          &&
  sys'.resources = sys.resources
}



pred show () {}
run show


-- no transitions outside of the sequence
fact every_transition_in_sequence
{
  Transition in set_up_seq/Seq::elems[]
}
-- Without this constraint sequences are created different transitions
fact exactly_one_sequence
{
  one Seq
}

assert SecureTrans
{
  all t: Transition | Secure[t.sys]
}
check SecureTrans for 3
check SecureTrans for 3 but exactly 1 Process
check SecureTrans for 4



assert no_loop{
        no m: Memory_Segment | not strongly_hierarchical[m]
}
check no_loop for 5 but exactly 1 Seq, exactly 1 SeqIdx
```

APPENDIX C: SEPARATION KERNEL RUNTIME FTLS IN ALLOY

```
/******************************************************************************

 A Least Privilege Model for Static Separation Kernels
  from  T.Levin, C.Irvine, T.Nguyen, CISR Tech Report NPS-CS-05-003
 October 2004
 (references to the Tables in the paper)

******************************************************************************/

module sep_kernel_run_time

-- library containing the dom function
open util/ternary as tern

-- For state transitions
open util/sequence[Transition] as run_time_seq


/******************************************************************************

The premilinary interface uses parameters such as resource_id and partition_id
to the interface functions.  In order to conform as much as possible to the
interace we have added these to the to Resource and Block.  The Alloy Analyzer
will place an integer identifier for each Resource, Block, etc. that it
created in a model.  Therefore, adding the Resource_ID and Partition_ID
constructs may not be entirely necessary, but creates for a stronger mapping.

disj is a native Alloy command meaning disjoint.  It used here to require two
elements to be distinct.

******************************************************************************/

-- Unique Resource identifier
sig Resource_ID {}

-- Every resource has a unique ID
fact
{
  all r1, r2: Resource |
    disj[r1,r2] => disj[r1.resource_id, r2.resource_id]
}
```

```
-- garuntees no resource_id's that do no participate in the universe
fact
{
  Resource_ID in Resource.resource_id
}

-- Retrieve the corresponding Resource
-- This function is not currently used
fun id_to_resource (i: Resource_ID): set Resource
{
  {

    r: Resource | r.resource_id = i
  }
}



-- Unique Block identifier
sig Partition_ID {}

-- Every block has a unique ID
fact
{
  all b1, b2: Block |
    disj[b1,b2] => disj[b1.partition_id, b2.partition_id]
}

-- garuntees no parition_id's that do no participate
fact
{
  Partition_ID in Block.partition_id
}

-- Retrieve the corresponding Block
-- This function is not currently used
fun id_to_block (i: Partition_ID): set Block
{
  {
  b: Block | b.partition_id = i
  }
}


/*****************************************************************************
```

```
The information in computer systems is often thought of generically as
resources.  The flow of information is then described as a matrix of what
resource is allowed to read and/or write to another resource.  This is
referred to as an access matrix.  A resource is anything in the system we wish
to consider, be it a file, a process, a user, a program, fireware, etc.  In
Alloy a 3-tuple or 3 item relation can be used to indicated the access matrix
like so, Resource -> Resource -> Mode  The 'abstract' keyword below indicates
that an item called Mode should not be created in the model.  The 'extends' and
'abstract' keywords are analogous to object oriented programming with a
heirarchy of parent and child classes.  The 'one' keyword means one and only
one item called RD and one and only one item called WT should be the model.

******************************************************************************/

-- R in the paper
sig Resource
{
-- Every Resource belongs to exactly one Block, no more, no less
  master: one Block,

-- Added to conform to the interface
  resource_id: one Resource_ID
}

-- access rights type, F in the paper
abstract sig Mode {}

-- read and write in the Paper
one sig RD, WT extends Mode {}


/******************************************************************************

A separation kernel partitions all the resources.  The separation kernel is
simplistic in that it only provides the isolation of resources into partitions
which we will call Blocks.  In the system every Resource will be assigned
exactly one and only one Block, thus the Blocks represent equivalence classes
of Resources.  Our separation kernel will be static, therefore there will be
no reassigning of Resources to different Blocks once the kernel is running and
it will not make sense to have empty Blocks.  It is not useful to have all
resources partitioned into a single Block so that case may be explicitly
excluded from the model in order to make it more interesting.

******************************************************************************/
```

```
-- B in the paper
sig Block
{
-- Added to conform to the interface
  partition_id: Partition_ID
}
-- each block has at least one resource, no empty blocks
{
  some r: Resource | r.master = this
}

-- a minimally useful kernel, this fact may be commented out
fact { #Block > 1 }


/******************************************************************************

The system will contain two matrices.  One to represent the flow of information
between blocks and one for the flow of information between resources.  The two
matrices are intended to orthogonal, that is, there may be flows allowed in one
that are not allowed in the other.  We will be interested in taking the partial
ordering and the trusted partial ordering of the block flow matrix.  To help
with the trusted partial ordering we create the structure below.

We can consider a read originating from BlockA to BlockB to be the same as a
write originating from BlockB to BlockA in terms of information flow.  We can
simplify either statement by saying information has flowed from BlockB to
BlockA.

Alloy has a built in command ^ (carrot) for constructing transitive closures.
The transitive closure is useful for seeing all blocks that information can
eventually flow into.

******************************************************************************/

-- Block-To-Block Flow Matrix, Table 1
sig BB{
  flow:        Block -> Block -> Mode,

-- those are secondary, derived from the flow
  basic_flow:  Block -> Block,
  FLOWS:       Block -> Block
}
{
```

```
-- definition of basic flow
  basic_flow =
  {
    a, b: Block |
      WT in flow[a][b] or
      RD in flow[b][a]
  }

-- FLOWS is a transitive closure of basic_flow
  FLOWS = ^ basic_flow

-- These two are for better visualizing and more interesting models

-- block always can access itself with Read/Write
--  all b: Block | #b.(b.flow) = #Mode

-- each access mode is represented at least once in the elements of flow
--    RD in Block.(Block.flow)
--    WT in Block.(Block.flow)
}


/*******************************************************************************

The word subject and object are typically used to indicate which resource is
the active entity performing the read and/or write on another passive resource.
In our model we use the Subject terminology and we allow one Subject to
operate on another Subject (ie. one process communicating to another process).
We drop the Object nomenclature and simply refer to a Subject accessing a
Resource (which may be another Subject).

*******************************************************************************/

sig Subject extends Resource {}


/*******************************************************************************

In order to simplify the modeling of reading and writing we simply declared a
memory independent of resources.  We leave processes and handels out of this
model.

*******************************************************************************/

sig Memory_Segment { }
```

```
/******************************************************************************

Our system is really rather simple.  It is two access matrices.  One is the
block flow (bb) and the other the least privilege matrix (sr_flow) which
constrains the block flow (bb).  Later, we will define a secure system to be a
system that only allows flows that are permitted by both matrices.  Technically
sr_flow is not required to be a subset of bb.  That is, it may allow flows that
are prohibited by bb, however, in a secure system those flows would never be
realized, because what is allowed is only the intersection of the two matrices.

A Block may contain any combination of Subjects and Resources.  Therefore, the
Block nomenclature is devoid of the Subject and Resource distinction.  In
order to better visualize and talk about the flow of information only Subjects
are allowed to read and/or write.  That is Resources that are not Subjects
should never initiate a flow.

Of course we are only interested in Resources and Blocks that are inside of our
System.  Sense all Resources belong to a Block, Alloy will not produce
Resources or Blocks that have nothing to do with each other.  In like manner we
need to instruct Alloy that every block participates in the System to prevent
Alloy from producing models with Resources and Blocks that are outside of the
System.

MM should be orthogonal to SR and BB, that is SR/BB is the policy of what is
allowed, and MM is what the program wants to do.  The security predicate will
say that a secure system allows only the program actions that conform to the
policy.

It is important to note that information is not required to flow into or out of
a Block.  However, a completely isolated Block would not be useful.

******************************************************************************/

sig System {
  resources: set Resource,

  bb: BB,
  sr_flow: Subject->resources->Mode,
  MM: Subject -> Resource -> Mode,

  -- This used in this model for modeling reading and writing.
  RM: resources -> Memory_Segment
}
```

```
{
-- Every Block participates in the System.
-- We start with an empty matrix and then we add blocks
  Block in Block.((bb.flow).Mode) + ((bb.flow).Mode).Block

-- sr_flow is non empty
  some sr_flow

/*
&&
-- This constraint is only to create interesting models.  It is recommend to
-- comment out this section to study the model the way it was intended.
-- A completely isolated Block is not useful therefore:
  -- Every block,
  all b1: blocks |
    -- has some other block,
    some b2: Block |
      -- that is not itself,
      disj[b1,b2]  &&
      (
        -- which it neither flows into,
        b2 in b1.flow or
        -- nor out of.
        b2 in flow.b1
      )
*/
}


/******************************************************************************

We are interested with a traditional confidentiality policy between the Blocks
that prohibits reading up or writing down.  The separation kernel itself only
isolates the Blocks and regulates the flow between them, but is devoid of
semantics labeling one Block greater than another.

However, we can define a predicate that will require the Blocks to be
partitioned in such a way that information is not allowed to flow circularly.
That is if information leaves BlockA, there is no transitive closure of the
flows that will lead back to BlockA. It is important to note that any two
Blocks are not required to be related.  For example, BlockA->BlockC,
BlockB->BlockC, but there is no flow between BlockA and BlockB and no
indication if one is greater than the other.  Therefore such a non-circular
flow of information is technically considered a partial ordering but not a
total ordering.  That is, not all the items in the set are comparable.
```

```
Mathematically a partial ordering (PO) is defined as a relation over a set that
is transitive and antisymmetric.  Antisymmetric means there is no circularity
in the relation.  Many authors extend a partial ordering to include reflexivity
which means every item in the set is comparable to itselfs.  In our situation
this may or may not be the case.  We include reflexivity in PO allowing it to
be commented out if not desired.

Note: A lattice is a partial ordering where every comparable pair of items
contain both a least upper bound (lub) and a greatest lower bound (glb), which
results in a universal upper and lower bound to the lattice.

******************************************************************************/

pred PO(bb: Block->Block){
  all  i,j,k: (bb.Block + Block.bb) |

-- reflexive
    (
      i->i in bb
    ) &&

-- antisymmetric
    (
      ((i->j in bb) && (j->i in bb)) => (i=j)
    ) &&

-- transitive
    (
      ((i->j in bb)  && (j->k in bb)) => (i->k in bb)
    )
}


/******************************************************************************

Overtime an item of information may no longer be sensitive and its sensitivity
may need to be downgraded.  To do so requires violating the partial ordering.
Therefore the notion of a Trusted Subject is introduced.  A Trusted Subject is
a Subject (ie. process) that has undergone rigorous analysis and is trusted not
to downgrade information other than the information it is intended to
downgrade.

******************************************************************************/
```

```
sig Trusted_Subject extends Subject {}


/******************************************************************************

A Trusted Subject is allowed to violate the partial ordering but it is not
required to violate the partial ordering.  Therefore, there may be flows in the
System that are the result of the Trusted Subjects and violate the partial
ordering.  The problem here is that we do not really have a way of identifying
which Subject, trusted or otherwise, caused a flow in bb, after all the
matrices are intended to be orthogonal.  However, the reverse may be possible.
We can identify the flows in sr_flow and require any flow that upsets the
partial ordering in bb to be a trusted subject.  However, this also requires us
to identify the partial ordering.

******************************************************************************/

pred TPO(sys: System){
  let Nontrusted_Subs_in_SR = dom[sys.sr_flow] - Trusted_Subject,

      Nontrusted_Block_Flow = { b1, b2: Block , m: Mode |
                 (
        some sub: Nontrusted_Subs_in_SR, r: Resource |
        (
          -- sub is a non-trusted subject in the subject part of sr_flow
          -- when combined with some resource and
          -- and the mode of Nontrusted_Block_Flow is also in sr_flow
          (sub -> r -> m) in sys.sr_flow and

          -- and the corresponding blocks of that subject and resource
          -- comprises the blocks of Nontrusted_Block_Flow
          b1 = sub.master and b2 = r.master
        )
      )
      } |

  -- The transitive closure of
  -- the intersection of the Nontrusted_Block_Flow and bb.flow
  -- with the Mode removed
  -- should be a partial ordering
  PO[ ^((Nontrusted_Block_Flow & sys.bb.flow).Mode) ]
}


/******************************************************************************
```

```
A secure system is a system that is a trusted partial ordering where only flows
in by both matrices (sr_flow and bb.flow) are allowed.

******************************************************************************/

pred Secure(sys: System)
{
  TPO[sys]  &&
  all sub: Subject, res: Resource, mod: Mode |
  (
    (sub -> res -> mod) in sys.MM

    =>

    (
      (sub -> res -> mod) in sys.sr_flow &&
      mod in sys.bb.flow[sub.master][res.master]
    )
  )
}


/******************************************************************************

The above defines a secure system and what it means to be in a secure state.
In the strictest sense the concept of transitioning from one state to another
does not exist in this definition because the system is static and either the
system is secure or it isn't.  The matrices do not change once brought into
existence.

In order to visualize transitions operation we create a Transition signature
that includes the current system along the with the last command executed, the
corresponding arguments, and the last error message.

This model represents the runtime phase of the system.

******************************************************************************/

-- List of commands in the preliminary interface.
abstract sig Command {}
one sig no_op_com,
    read_com,
    write_com
extends Command {}
```

```
-- An error for each command
abstract sig Error { }
one sig no_err,
    read_err,
    write_err
extends Error {}

sig Transition
{
  error_message: Error,
  last_command:  Command,
  arguments:     set univ,

  sys: System
}


/******************************************************************************

We begin the ordering of Transitions (states) with a secure System and no error
messages nor commands having been executed.

Then we define a transition as having unchanged matrices and either a read or
a write having occurred.  In the transition the security of the new System is
not stated.  This is left for an assertion to verify that no transition will
lead to an unsecured System.

******************************************************************************/

fact init_run_time
{
  let t = Seq::run_time_seq/first[] |

    no_err    = t.error_message &&
    no_op_com = t.last_command  &&
    no t.arguments              &&

    Secure[t.sys]
}


fact trans_run_time
{
  -- This is for more interesting models
```

```
--set_up_seq/allExistNoDuplicates[] &&

-- For all seqence indices except the first one
all sqidx: Seq::run_time_seq/inds[] - run_time_seq/firstIdx[] |
{

 let t = run_time_seq/Seq::at[ run_time_seq/prev[sqidx] ],
     t'= run_time_seq/Seq::at[sqidx]
  |

  -- To prevent a tranisition that is the same command, same args,
  -- same error and same system from occuring next to each other
  --disj[t'.sys, t.sys] &&

  some s: System,
       -- given some resource and subject in the system
       res: (t.sys).resources, sub: (Subject & res),
       -- and their corresponding memory_segments
       segR: res.(t.sys.RM), segS: sub.(t.sys.RM) |
  {

    -- either a read
    (
      (
        read[t.sys, s, res, segR, sub, segS]   &&
        Secure[s]
      )
      =>   (
             t'.sys           = s            &&
             t'.last_command  = read_com     &&
             t'.arguments     = res + segR +
                                sub + segS   &&
             t'.error_message = no_err
           )
      else (
             t'.sys           = t.sys             &&
             t'.last_command  = read_com     &&
             t'.arguments     = res + segR +
                                sub + segS   &&
             t'.error_message = read_err
           )
    )

    ||
      -- or a write occurred
      (
        (
          write[t.sys, s, res, segR, sub, segS] &&
          Secure[s]
        )
        =>   (
               t'.sys           = s            &&
               t'.last_command  = read_com     &&
               t'.arguments     = res + segR +
                                  sub + segS   &&
               t'.error_message = no_err
             )
        else (
               t'.sys           = t.sys            &&
               t'.last_command  = write_com    &&
               t'.arguments     = res + segR +
                                  sub + segS   &&
               t'.error_message = write_err
             )
      )

    }--some

  }--all
}


/*******************************************************************************

Memory modificaiton represented in the read and write commands.

*******************************************************************************/

pred read(sys: System,   sys': System,
          res: Resource, segR: Memory_Segment,
          sub: Subject,  segS: Memory_Segment)
{
  -- then memory modification occurs
  sys'.RM = ((sys.RM)-(sub->segS)) + (sub->segR) &&

  sys'.bb.flow = sys.bb.flow  &&
  sys'.sr_flow = sys.sr_flow  &&
  sys'.MM      = sys.MM
}
```

```
pred write(sys: System,   sys': System,
           res: Resource, segR: Memory_Segment,
           sub: Subject,  segS: Memory_Segment)
{
  -- then memory modification occurs
  sys'.RM = ((sys.RM)-(res->segR)) + (res->segS)  &&

  sys'.bb.flow = sys.bb.flow  &&
  sys'.sr_flow = sys.sr_flow  &&
  sys'.MM      = sys.MM
}


pred show () {}
run show


-- no transitions outside of the sequence
fact every_transition_in_sequence
{
  Transition in run_time_seq/Seq::elems[]
}
-- Without this constraint sequences are created different transitions
fact exactly_one_sequence
{
  one Seq
}

assert SecureTrans { all t: Transition | Secure[t.sys] }
check SecureTrans for 3
check SecureTrans for 4
```

APPENDIX D: MORPHISM EXAMPLE IN ALLOY

```
module morphism_demo

/******************************************************************************

Specware 4.1 Tutorial
Chapter 1.3.4 Morphisms pag3 7

As another, less trivial example, consider a spec for natural numbers that also
includes an op plus and an op times, both of type Nat * Nat -> Nat. (The
construct “*” builds the cartesian product of two types: in a model, A * B
denotes the cartesian product of the set denoted by A and the set denoted by
B.) The spec also contains axioms that define plus and times to be addition
and multiplication. Now, consider another spec consisting of a type X, an op f
of type X * X -> X, and an axiom stating that f is commutative:
fa(x,y) f(x,y) = f(y,x)

******************************************************************************/



/******************************************************************************

Source Specification

/******************************************************************************/

one sig Nat {
  rel: Nat->Nat->Nat
}

pred op (n1, n2, n3: Nat) {
  (n1->n2->n3) in (Nat.rel)
}

fact communative {
  all n1, n2, n3: Nat |
    op[n1, n2, n3] <=> op[n2, n1, n3]
}


/******************************************************************************
```

```
Target Specification

/*****************************************************************************/

sig X extends Nat {}

pred opx(x1, x2, x3: X) {
  op[x1, x2, x3]
  -- further operation definition would go here
}

assert morphism{
  all x1, x2, x3: X |
    opx[x1, x2, x3] <=> opx[x2, x1, x3]
}
check morphism


pred show() {}
run show
```

LIST OF REFERENCES

[Com06] _. *Common CriteriaDocumentation*. <http://www.commoncriteriaportal.org/> August 2005

[Bel73] Bell, D.E., and LaPadula, L.J. *Secure computer systems: mathematical foundations and model*. M74-244, The MITRE Corp., Bedford, Mass, May 1973.

[Bis03] Bishop, M. *Computer Security - Art and Science*. First edn. Addison Wesley, 2003.

[Bow03] Bowen, Jonathan. *Formal Specification and Documentation using Z: A Case Study Approach*. International Thomson Computer Press, 1996.

[Dec06] DeCloss, Daniel. *An Analysis Of Specware And Its Usefulness In The Verification Of High Assurance Systems*. Master's Thesis, Naval Postgraduate School. June 2006

[Has04] Hashii, B. *Lessons Learned Using Alloy to Formally Specify MLS-PCA Trusted Security Architecture*. Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering (FMSE'04), Washington DC, USA, October 2004, pp. 86-95

[Irv04] Irvine, C. E., Levin, T. E., Nguyen, T. D., and Dinolt, G. W. *The Trusted Computing Exemplar*. Proceedings of the 2004 IEEE Systems, Man and Cybernetics Information Workshop, West Point, NY, June 2004, pp. 109-115.

[Jac01] Jackson, D., Shlyakhter, I., Sridharan, M. *A Micromodularity Mechanism*. Proceedings of the ACM SIGSOFT Conference on Automated Software Engineering (ASE), November 2001

[Jac06] Jackson, Daniel. *Software Abstractions*, The MIT Press. April 2006

[Lam71] Lampson, B. W. *Protection*. In Proceedings 5th Princeton Conference on Information Sciences. Princeton, NJ, 1971. Reprinted in Operating Systems Reviews, 8(1): 18-24, 1974.

[Lev04] Levin, T. E., Irvine, C. E., and Nguyen T. D. *A Least Privilege Model for Static Separation Kernels*. NPS-CS-05-003, Naval Postgraduate School, October 2004

[Mil89] Millen, J. K. *Finite-State Noiseless Cover Channels.* Second IEEE Computer Security Foundations Workshop (CSFW), Franconia, New Hampshire, June, 1989, pp. 81-86.

[Rus81] Rushby, John. *Design And Verification Of Secure Systems*. ACM Operating Systems Review, 15(5), 1981.

[Sal73] Saltzer, J. H., and Schroeder, M. D. *The Protection of Information in Operating Systems*. Proceedings of the IEEE, 63(0): 1278-1308, 1975.

[Spe04] _. *Specware Documentation*. <http://www.specware.org/doc.html> September 2004

[Ubh03] Ubhayakar, Sonali. *Evaluation of Program Specification and Verification Systems*. Master's Thesis, Naval Postgraduate School. June 2003

[Wei03] Weissman, Clark. *MLS-PCA: A High Assurance Security Architecture for Future Avionics*. Proceedings of the 19th Annual Computer Security Applications Conference, 2003.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Dr. Mikhail Auguston
   Naval Postgraduate School
   Monterey, California

4. Timothy Levin
   Naval Postgraduate School
   Monterey, California

5. David Phelps
   Naval Postgraduate School
   Monterey, California